Яндекс

Kotlin Muptiplatform: редьюсим сложность

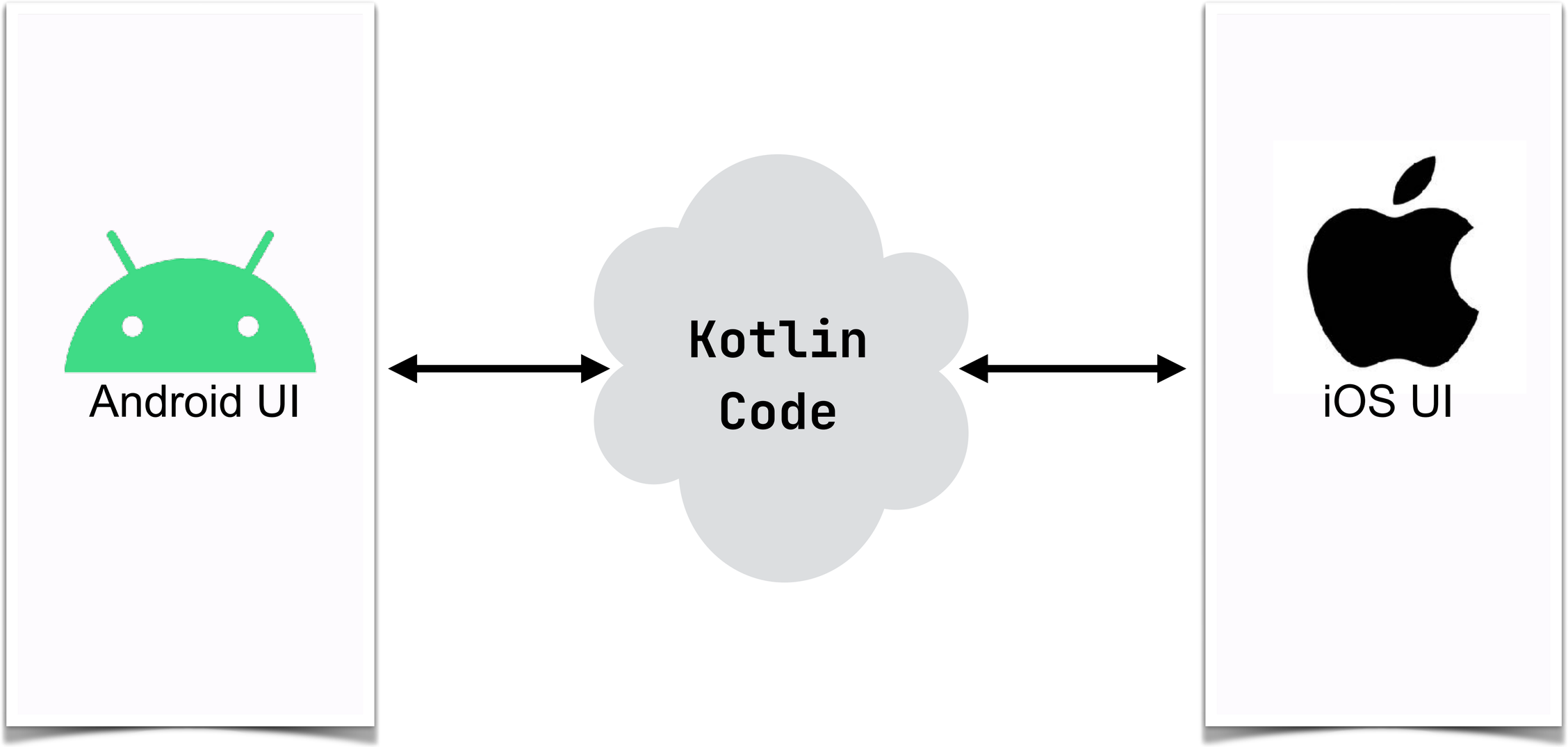
Денис Загаевский, Senior Android Developer an Yandex Maps

zagayevsky@yandex-team.ru

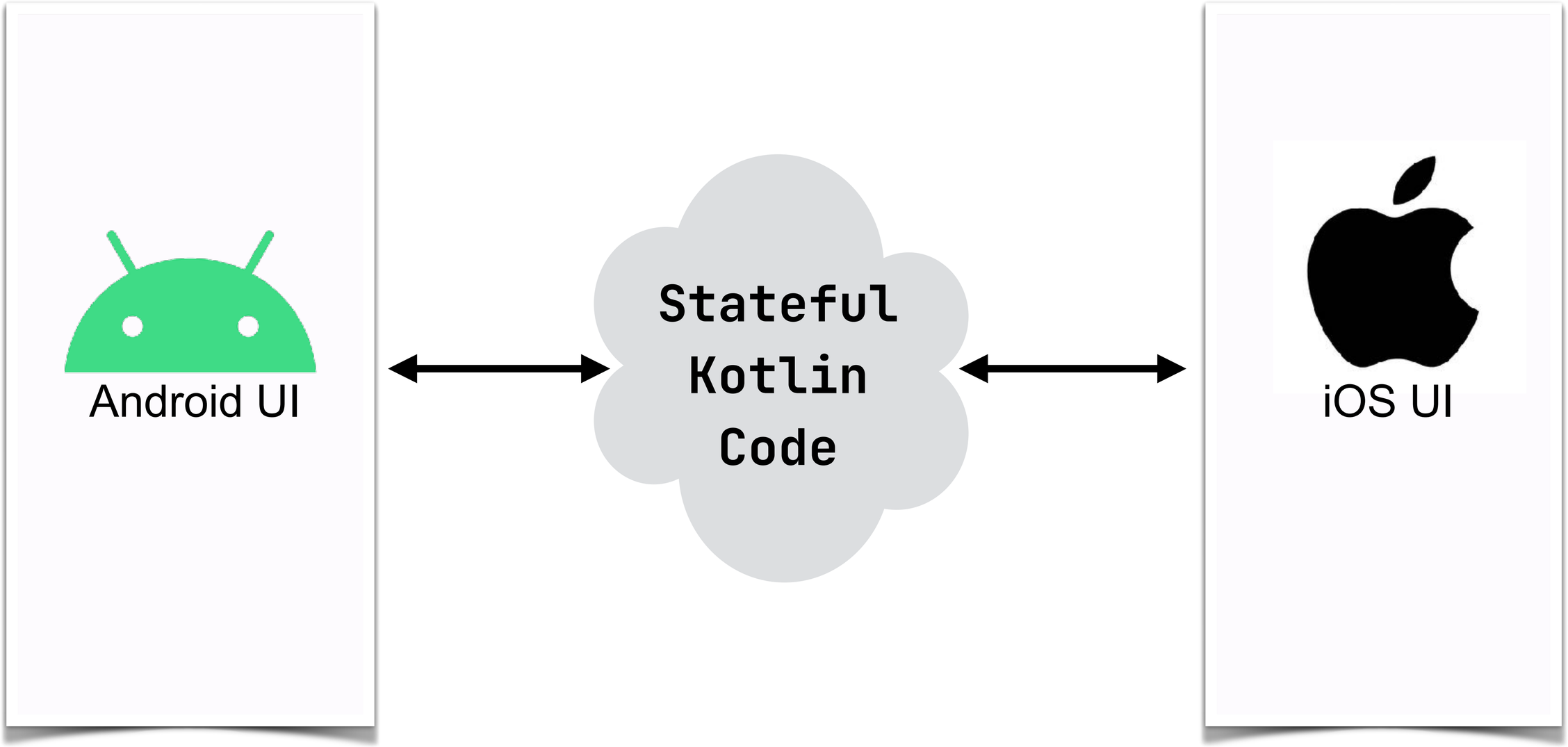
How to reduce complexity?

- Problem definition
- Basic architecture overview
- Architecture problems solution
- Modules decomposition

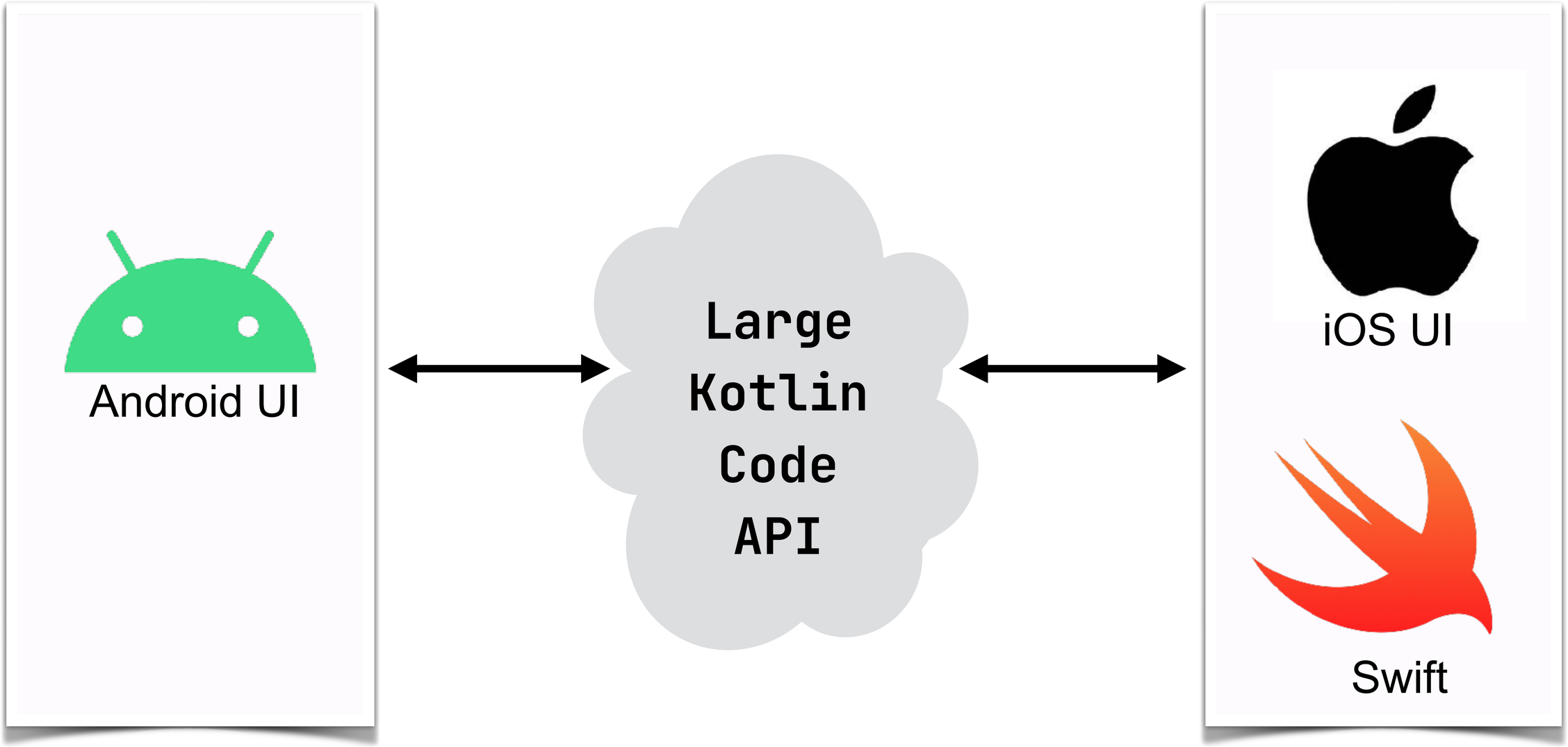
Problem definition — two Platforms



Problem definition — state management



Problem definition — visibility problems



Architecture

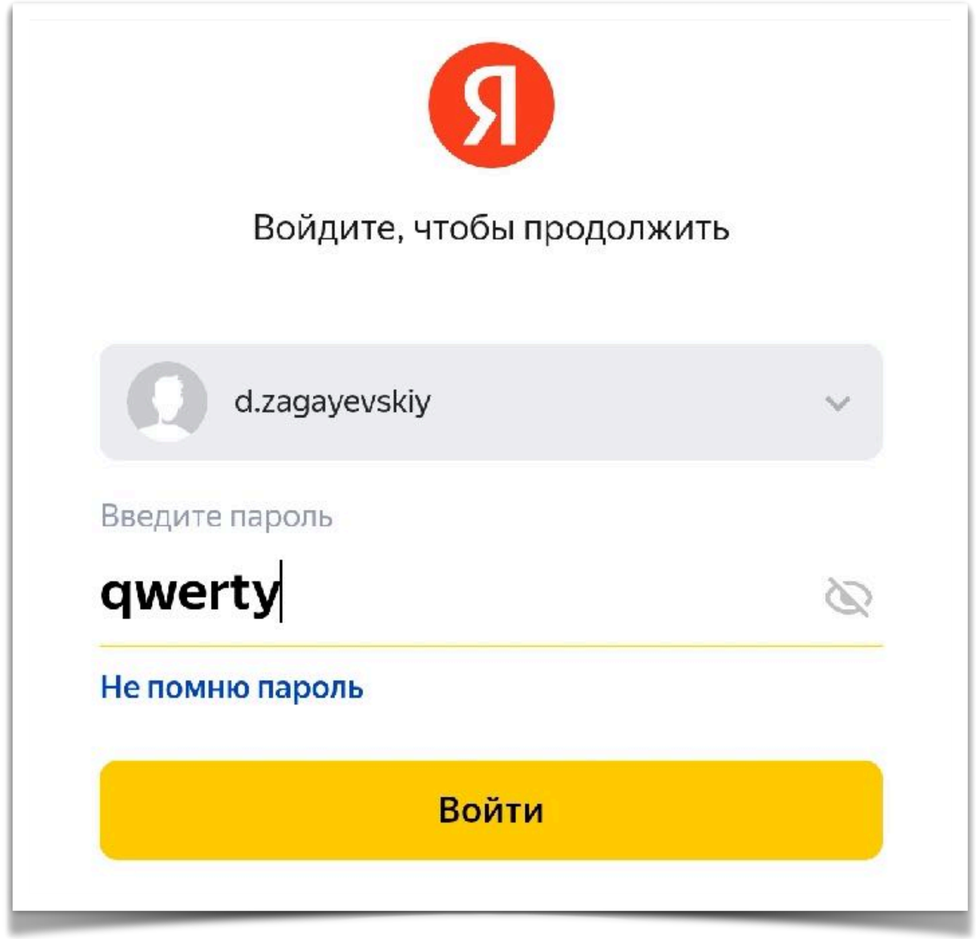
Redux

Easy state management

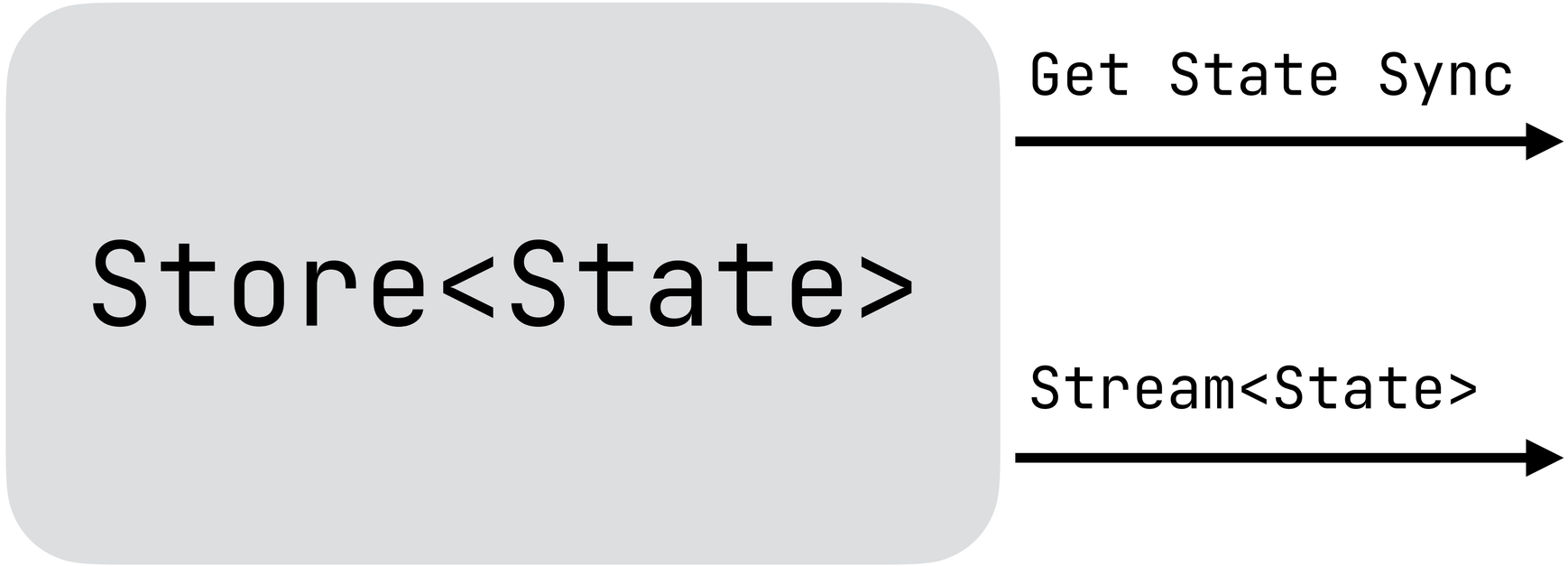
State

```
data class State(
    val username: String,
    val password: String?,
    val passwordHidden: Boolean,
    val imageUrl: String?,
)
```

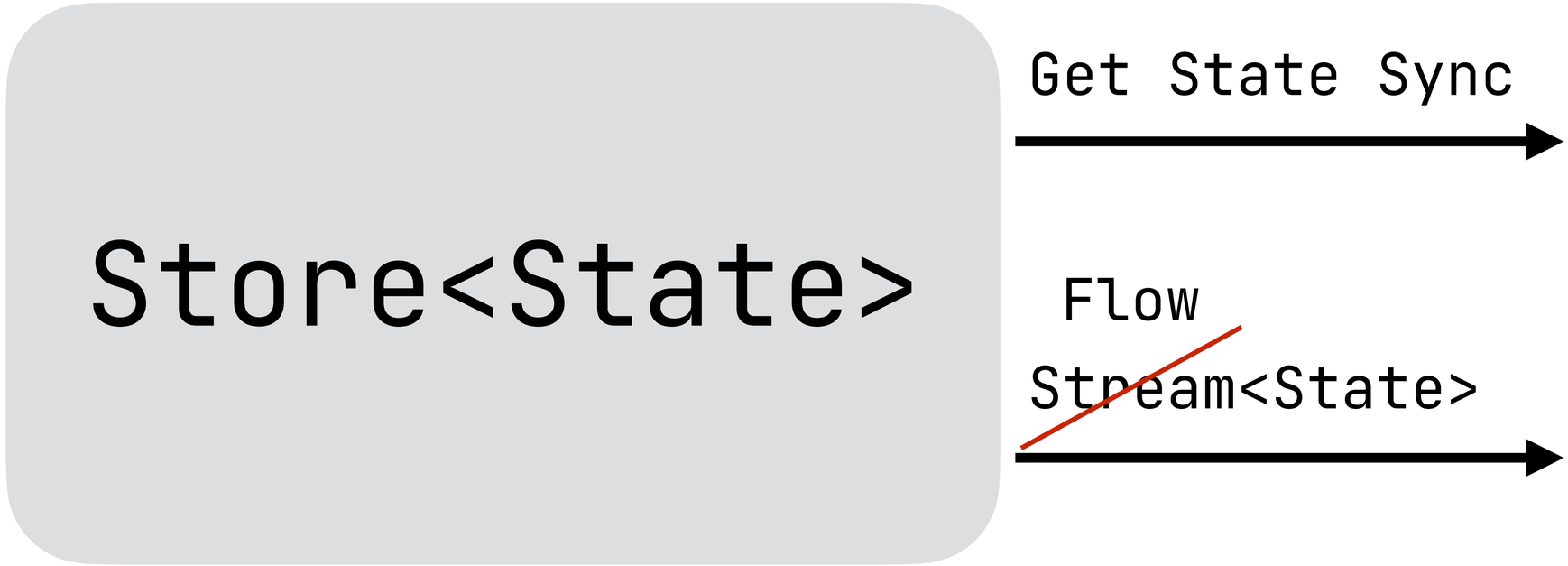
Store<State>



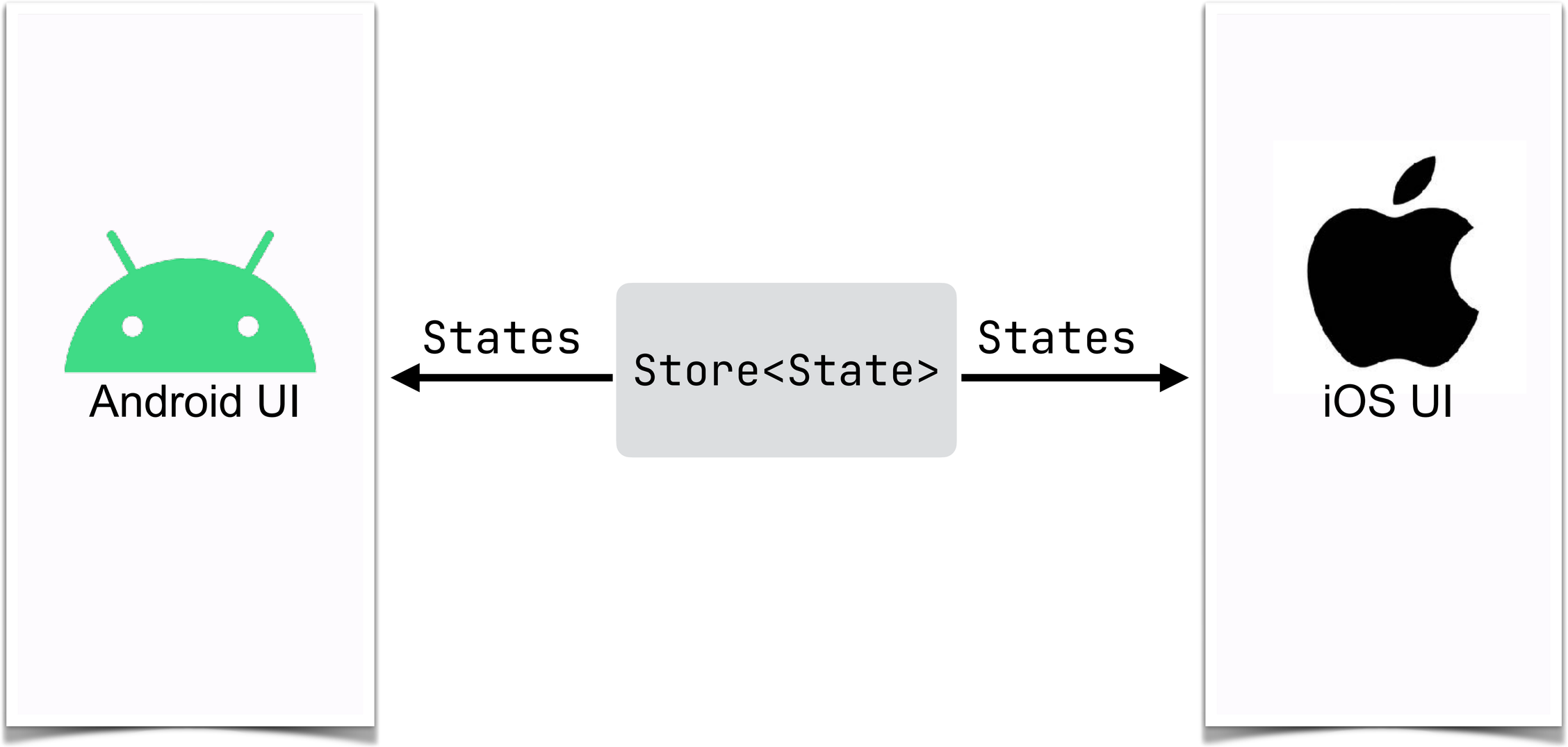
Store output



Store output



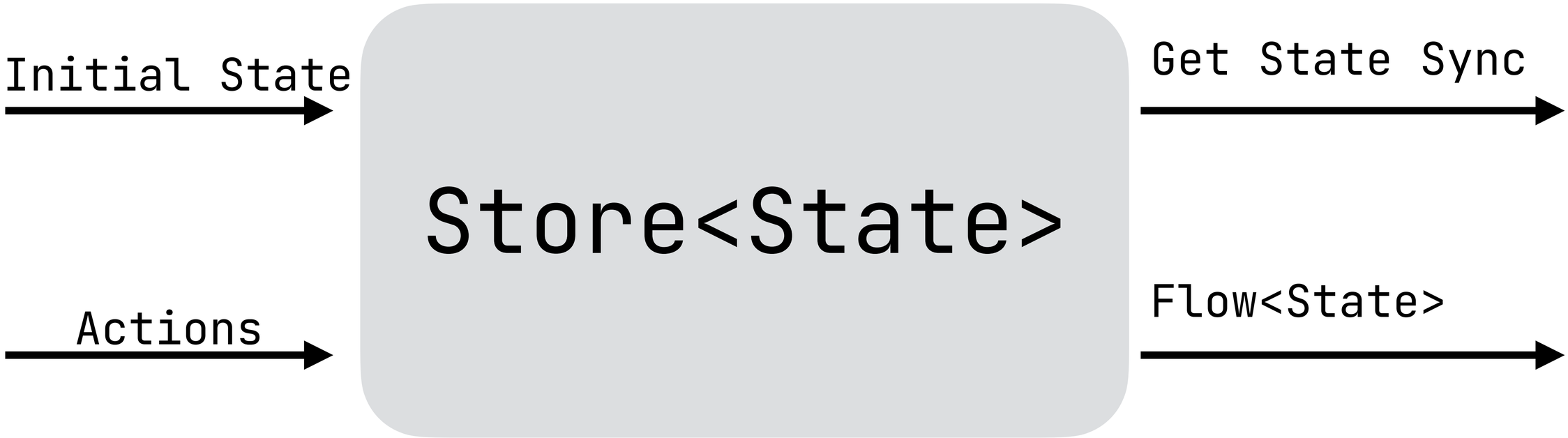
UI rendered from Store output



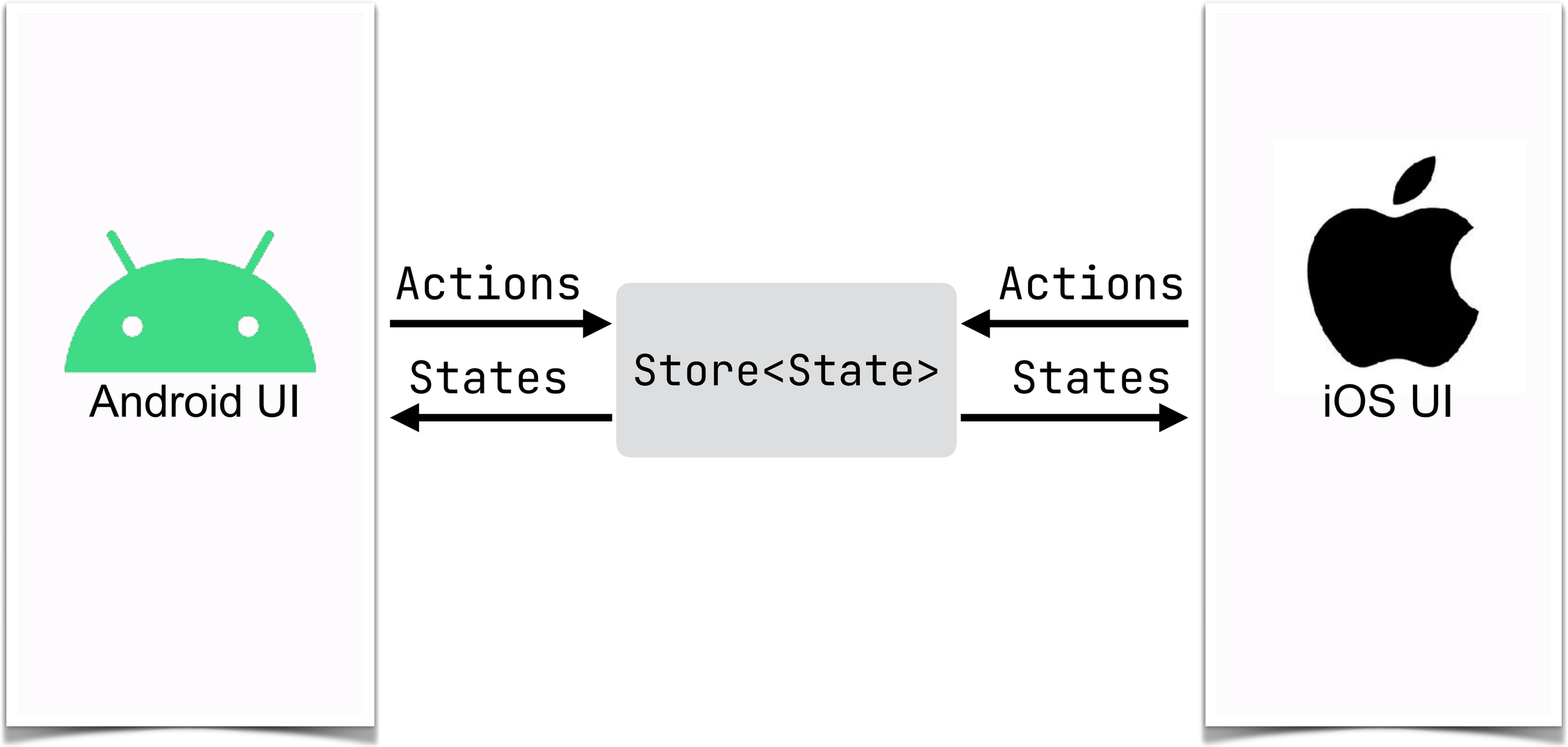
UI rendered from Store output


```
class MyFragment: Fragment() {
    @Inject
    internal lateinit var store: Store<MyState>

    override fun onViewCreated(…) {
        store.states()
            .onEach { state → render(state) }
            .launchIn(mainScope)
    }
    private fun render(state: MyState) { … }
}
```

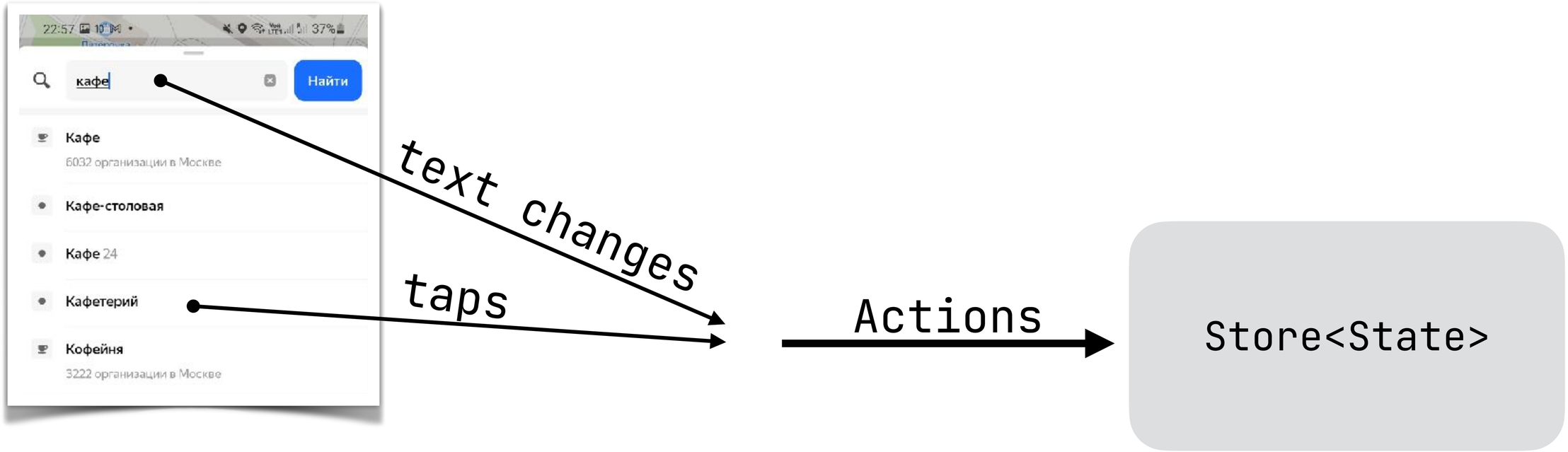
Store input


Initial State
Actions
Store<State>
Get State Sync
Flow<State>

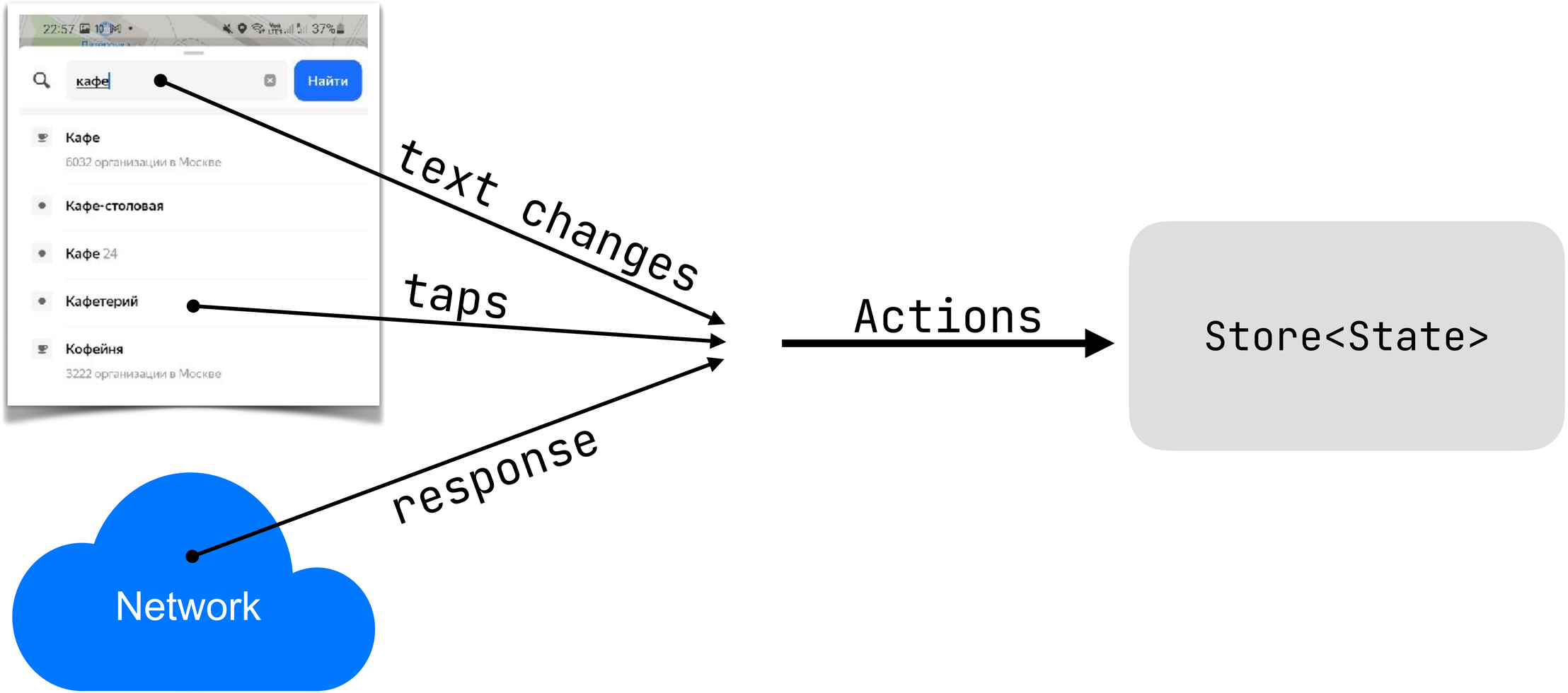
UI rendered from Store output



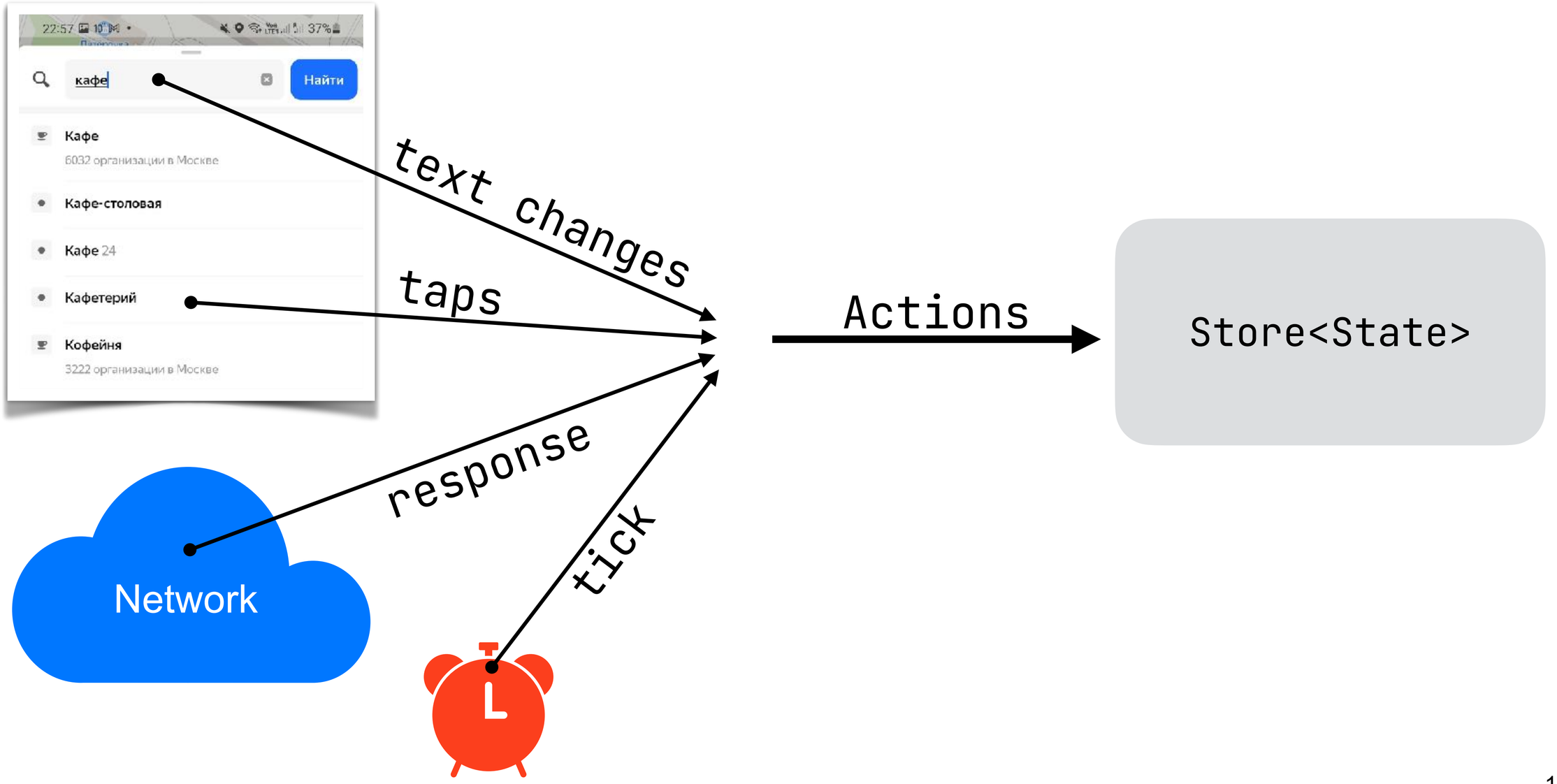
Actions — UI


22:57
37%
кафе
Найти
Кафе
6032 организации в Москве
Кафе-столовая
Кафе 24
Кафетерий
Кофейня
3222 организации в Москве
text changes
taps
Actions
Store<State>

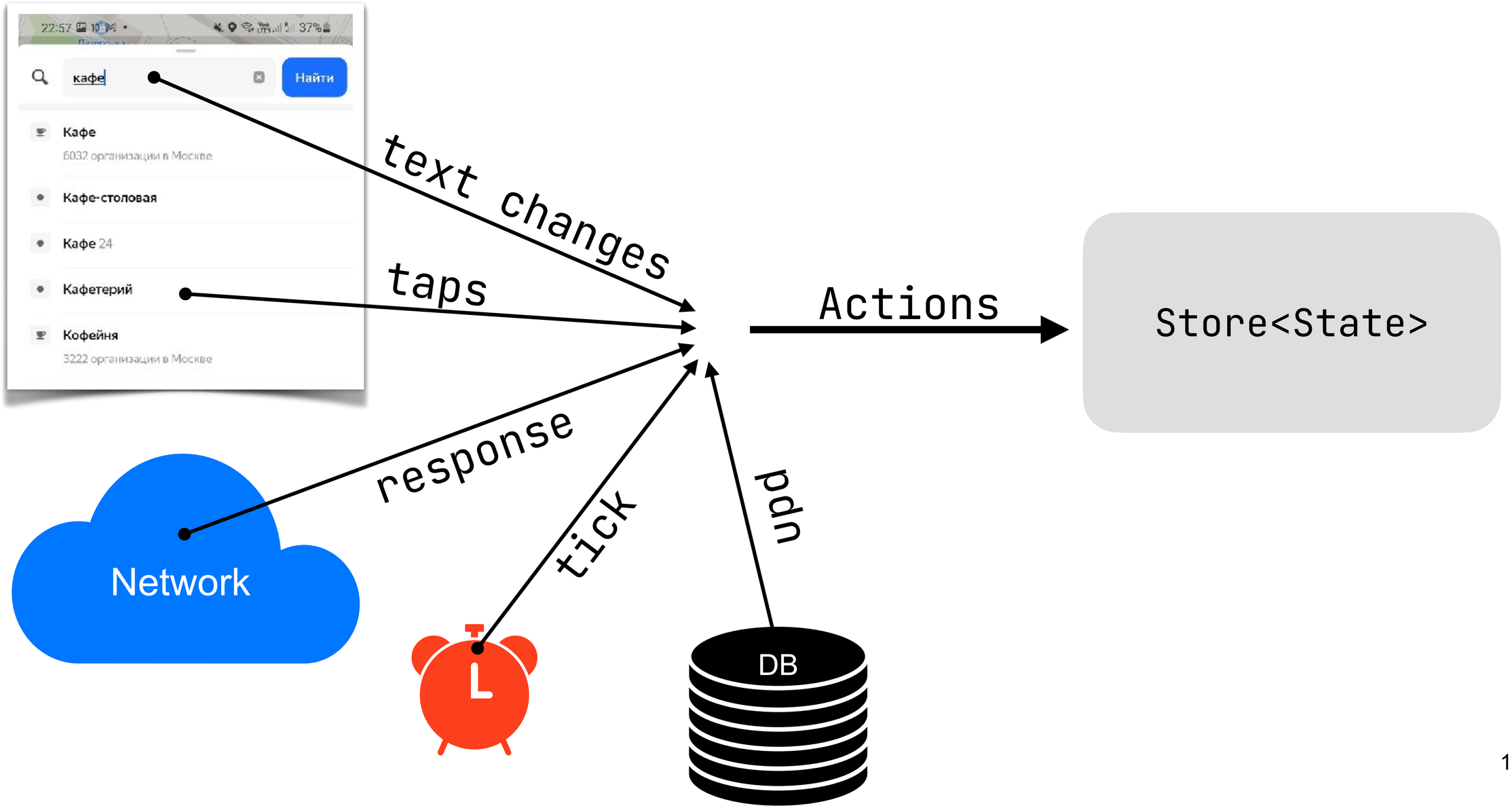
Actions — UI, Network



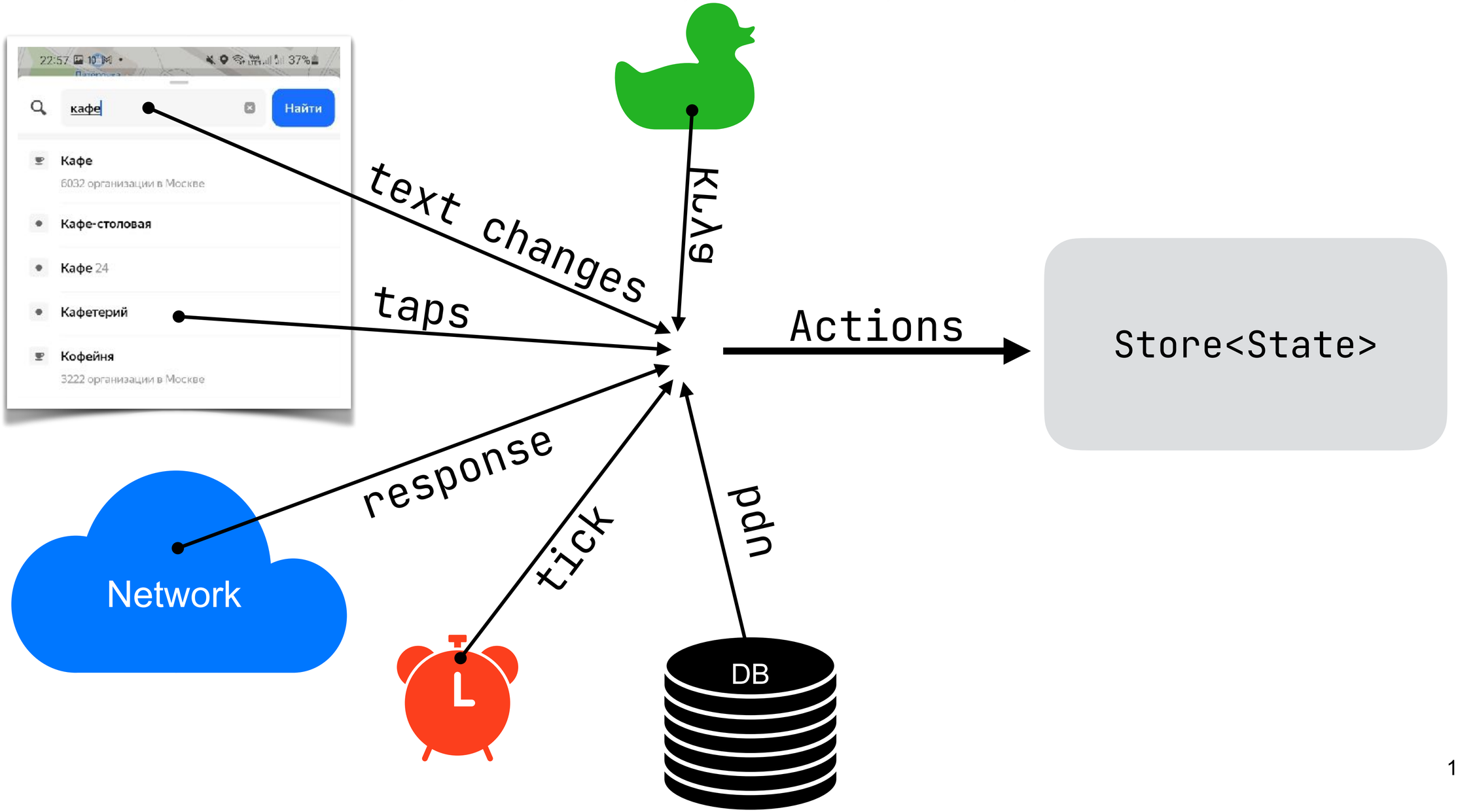
Actions — UI, Network, Timer



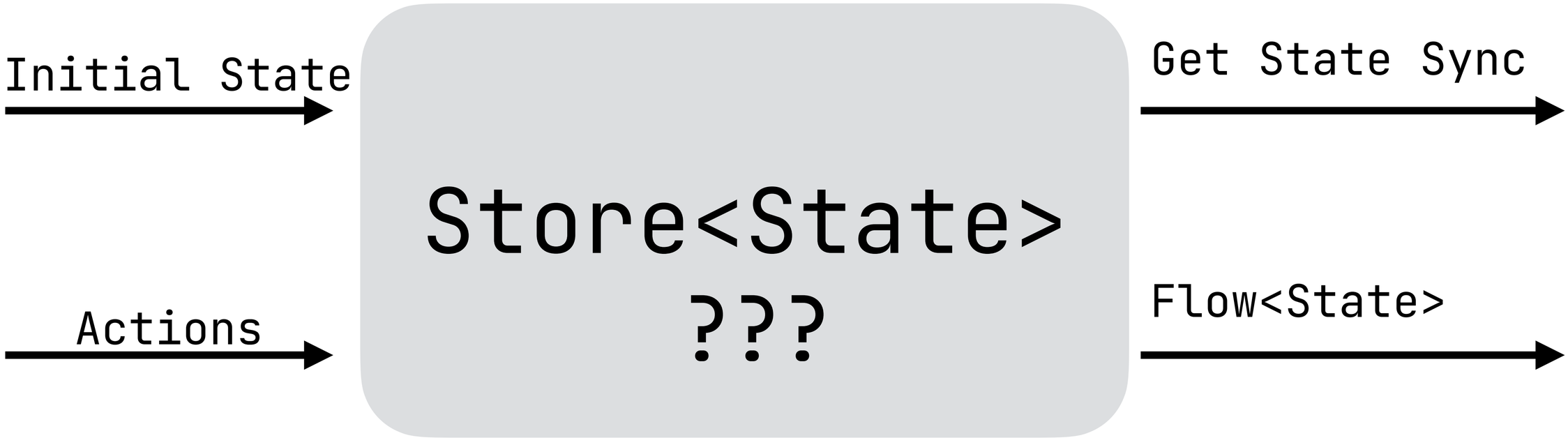
Actions — UI, Network, Timer, DB



Actions — UI, Network, Timer, DB, Whatever…



Store magic


Initial State
Actions
Store<State>
???
Get State Sync
Flow<State>

Reduce

```
fun reduce(state: State, action: Action): State
```

Reduce

```
fun reduce(state: State, action: Action): State

    Business logic
```

Reduce

```
fun reduce(state: State, action: Action): State
```

> Business logic

> Pure fun

Reduce example

```
data class CounterState(
    val name: String,
    val count: Int,
)

object Inc : Action
object Dec : Action
class IncBy(val value: Int): Action

fun reduce(state: CounterState, action: Action): CounterState = state.copy(
    count = reduceCount(state.count, action),
)

private fun reduceCount(count: Int, action: Action) = when(action) {
    Inc → count + 1
    Dec → count - 1
    is IncBy → count + action.value
    else → count
}
```

Reduce example

```
data class CounterState(
    val name: String,
    val count: Int,
)

object Inc : Action
object Dec : Action
class IncBy(val value: Int): Action

fun reduce(state: CounterState, action: Action): CounterState = state.copy(
    count = reduceCount(state.count, action),
)

private fun reduceCount(count: Int, action: Action) = when(action) {
    Inc -> count + 1
    Dec -> count - 1
    is IncBy -> count + action.value
    else -> count
}
```

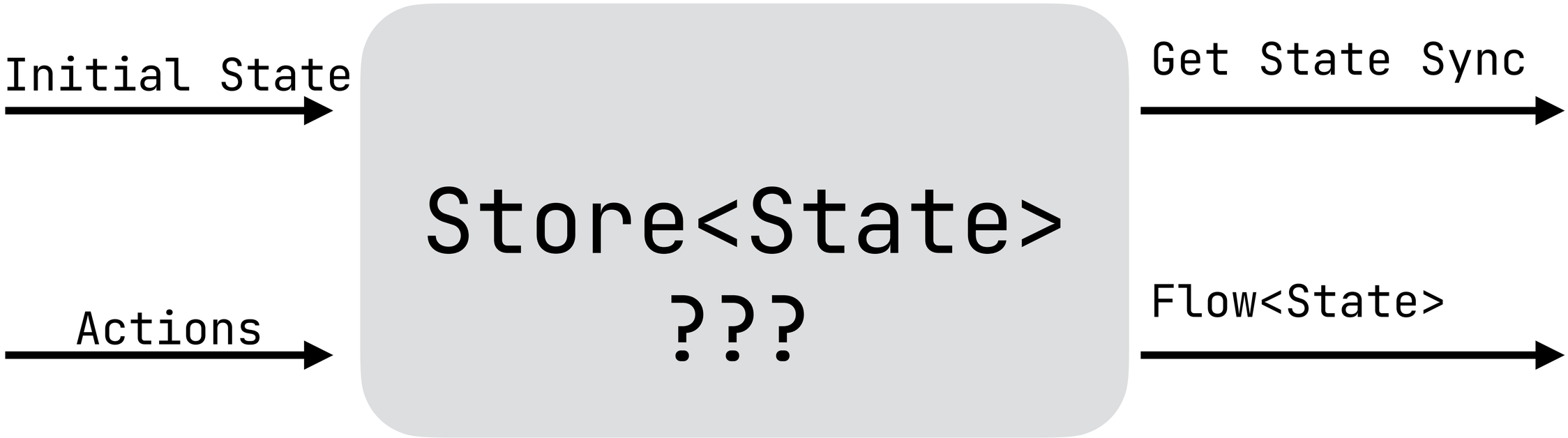
Reduce example

```
data class CounterState(
    val name: String,
    val count: Int,
)

object Inc : Action
object Dec : Action
class IncBy(val value: Int): Action

fun reduce(state: CounterState, action: Action): CounterState = state.copy(
    count = reduceCount(state.count, action),
)

private fun reduceCount(count: Int, action: Action) = when(action) {
    Inc -> count + 1
    Dec -> count - 1
    is IncBy -> count + action.value
    else -> count
}
```

Reduce example

```
data class CounterState(
    val name: String,
    val count: Int,
)

object Inc : Action
object Dec : Action
class IncBy(val value: Int): Action

fun reduce(state: CounterState, action: Action): CounterState = state.copy(
    count = reduceCount(state.count, action),
)

private fun reduceCount(count: Int, action: Action) = when(action) {
    Inc → count + 1
    Dec → count - 1
    is IncBy → count + action.value
    else → count
}
```

Store magic


Initial State
Actions
Store<State>
???
Get State Sync
Flow<State>

Store magic

```
class Store<State : Any>(
    initialState: State,
    private val reduce: (State, Action) → State,
) {
    private val statesSubject = MutableStateFlow(initialState)
    private val actionsSubject = MutableSharedFlow<Action>(…)
    private val scope: CoroutineScope = MainScope()

    init {
        actionsSubject.onEach { action →
            statesSubject.emit(reduce(currentState, action))
        }.launchIn(scope)
    }
…
```

Store magic

```
class Store<State : Any>(
    initialState: State,
    private val reduce: (State, Action) → State,
) {
    private val statesSubject = MutableStateFlow(initialState)
    private val actionsSubject = MutableSharedFlow<Action>(…)
    private val scope: CoroutineScope = MainScope()

    init {
        actionsSubject.onEach { action →
            statesSubject.emit(reduce(currentState, action))
        }.launchIn(scope)
    }
…
```

Store magic

```
class Store<State : Any>(
    initialState: State,
    private val reduce: (State, Action) → State,
) {
    private val statesSubject = MutableStateFlow(initialState)
    private val actionsSubject = MutableSharedFlow<Action>(…)
    private val scope: CoroutineScope = MainScope()

    init {
        actionsSubject.onEach { action →
            statesSubject.emit(reduce(currentState, action))
        }.launchIn(scope)
    }
…
```

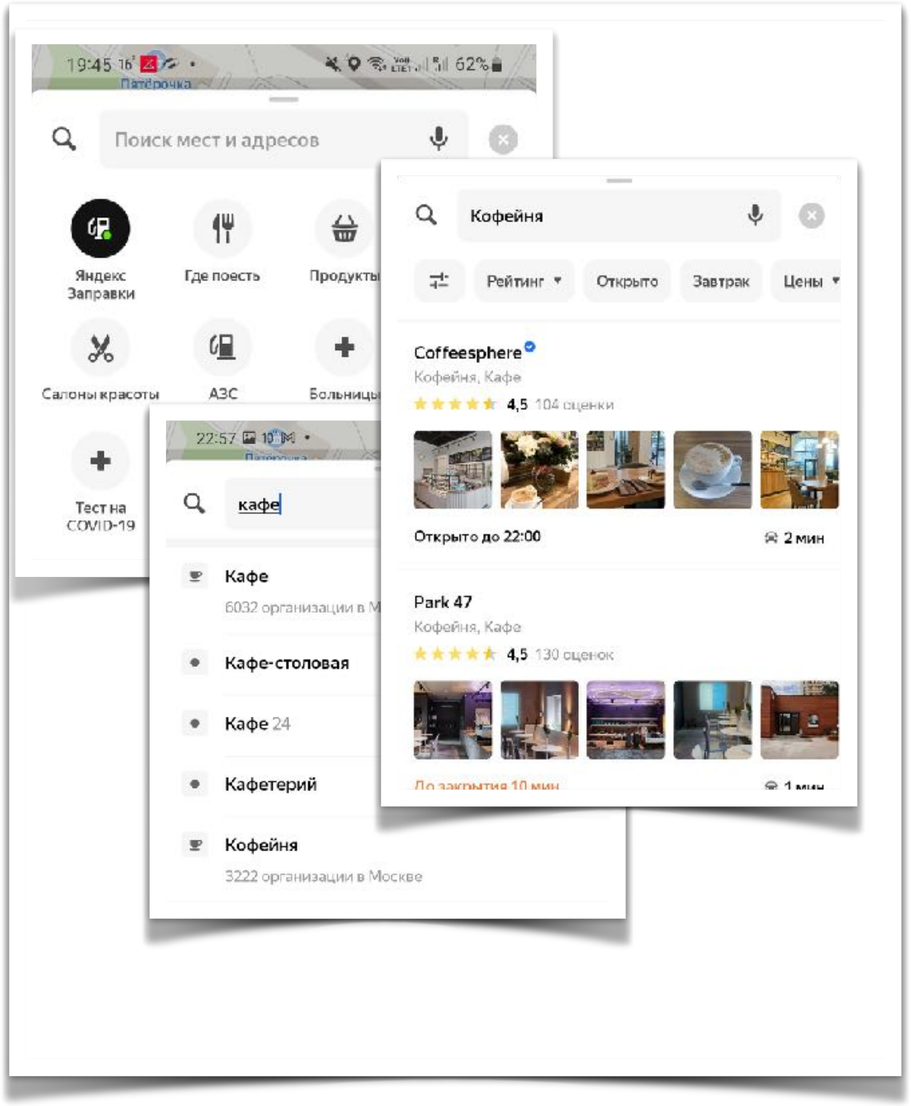
Store magic

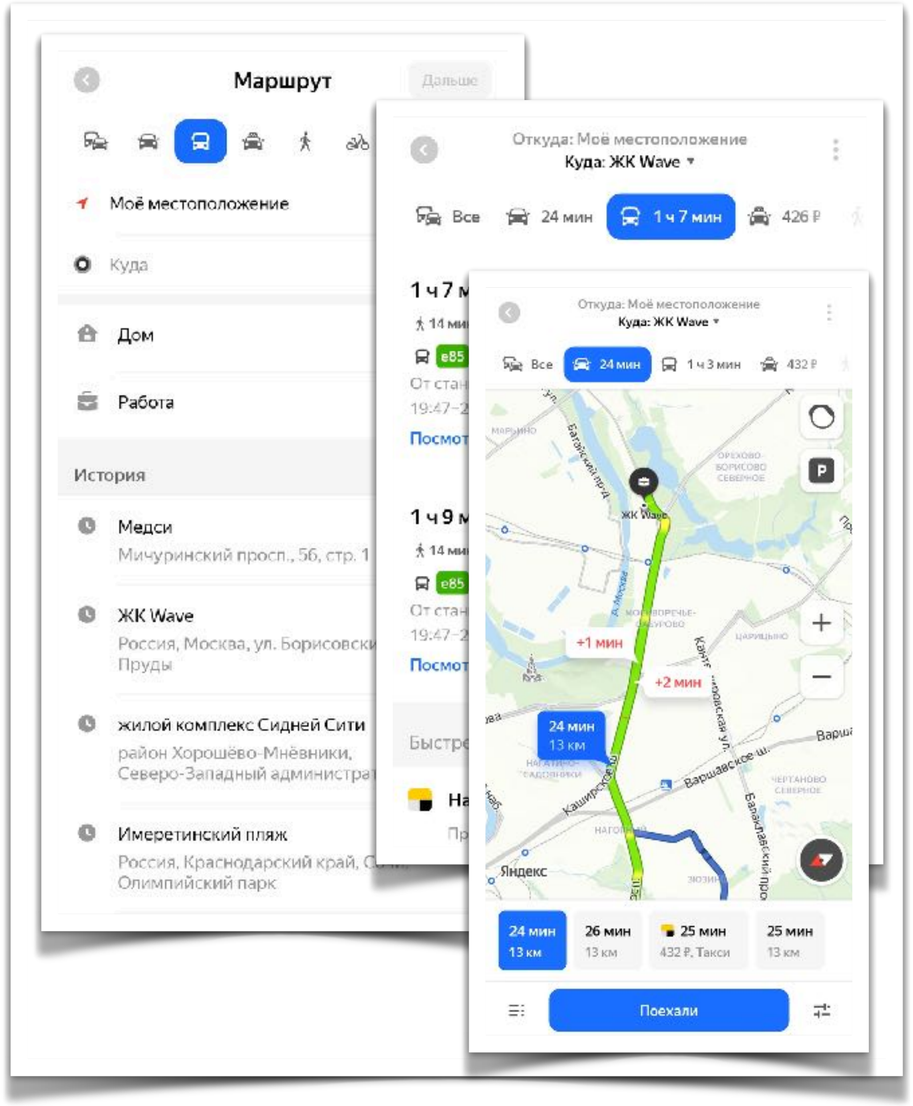
```
class Store<State : Any>(
    initialState: State,
    private val reduce: (State, Action) → State,
) {
    private val statesSubject = MutableStateFlow(initialState)
    private val actionsSubject = MutableSharedFlow<Action>(…)
    private val scope: CoroutineScope = MainScope()

    init {
        actionsSubject.onEach { action →
            statesSubject.emit(reduce(currentState, action))
        }.launchIn(scope)
    }
…
```

Store public interface

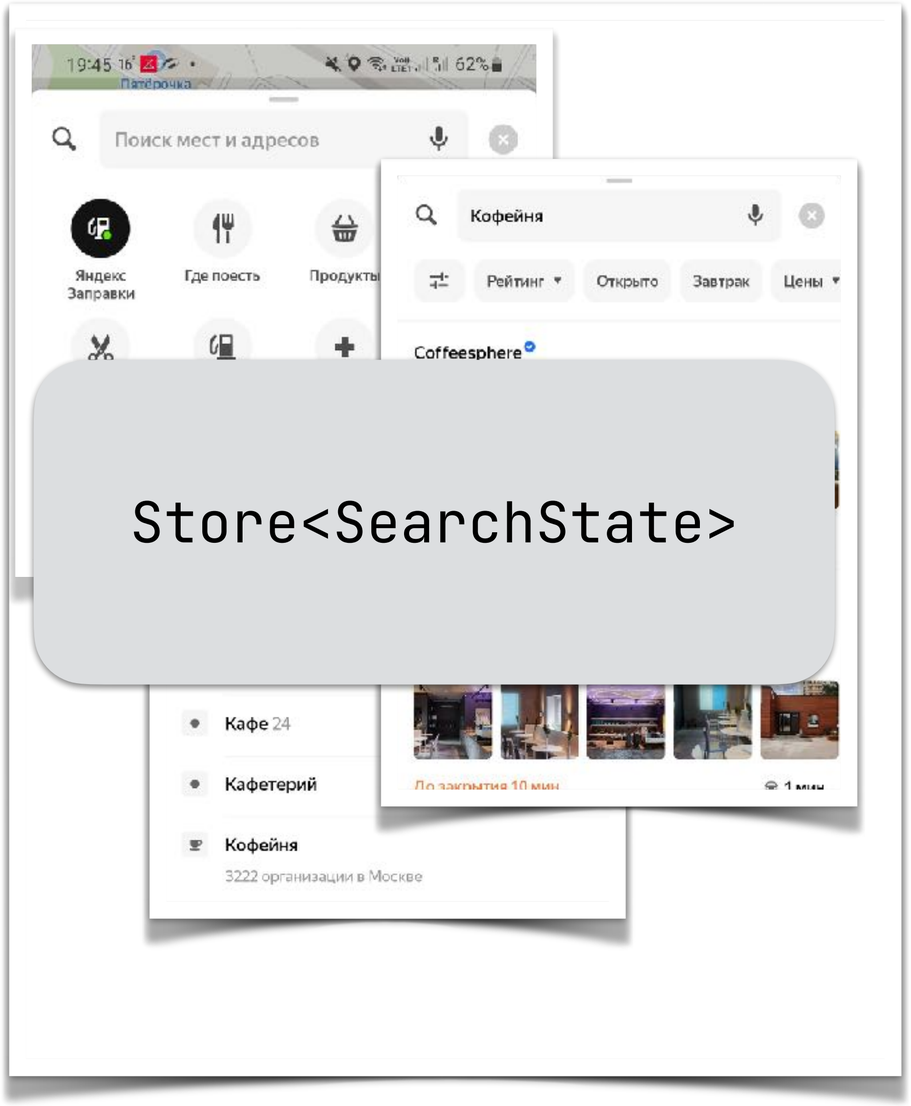
```
class Store<State : Any>(
    initialState: State,
    private val reduce: (State, Action) → State,
) {
    …
    val currentState: State
        get() = statesSubject.value

    fun states(): Flow<State> = statesSubject

    fun dispatch(action: Action) {
        scope.launch { actionsSubject.emit(action) }
    }
}
```

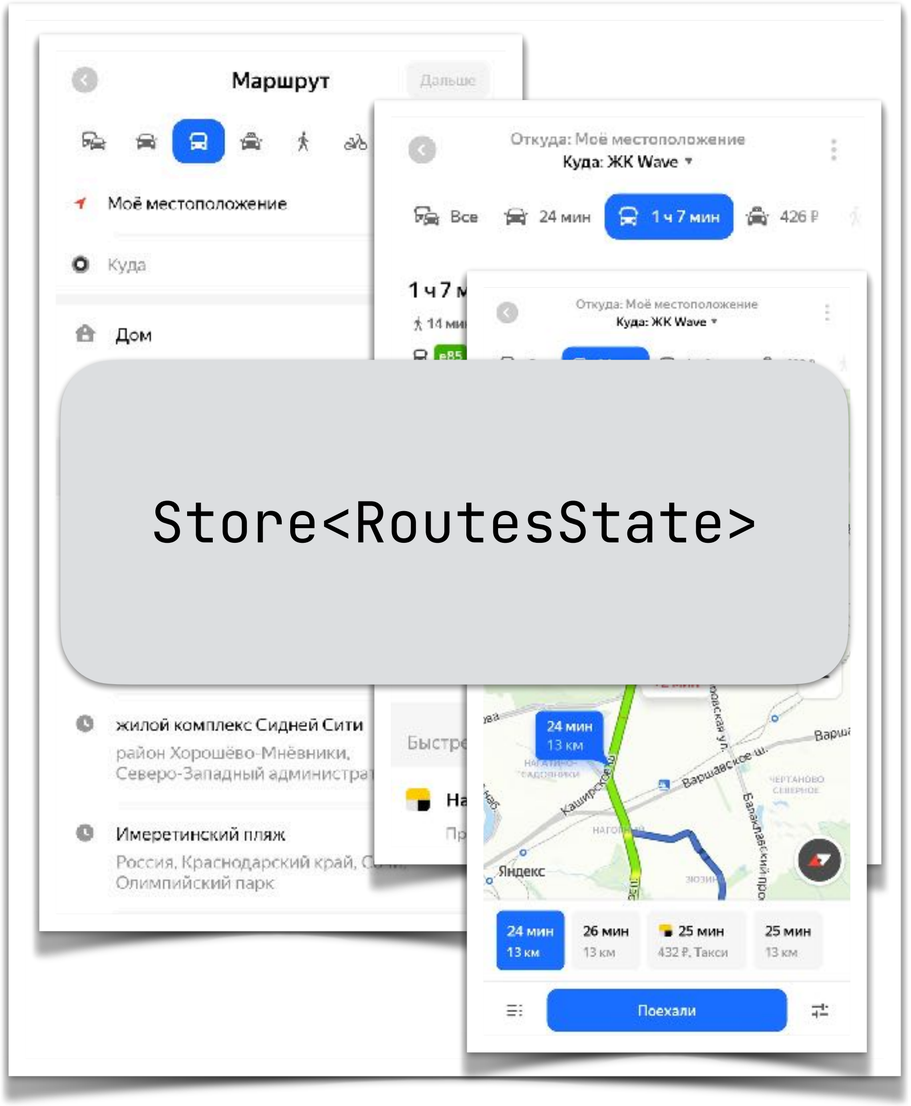
One app — multiple Stores



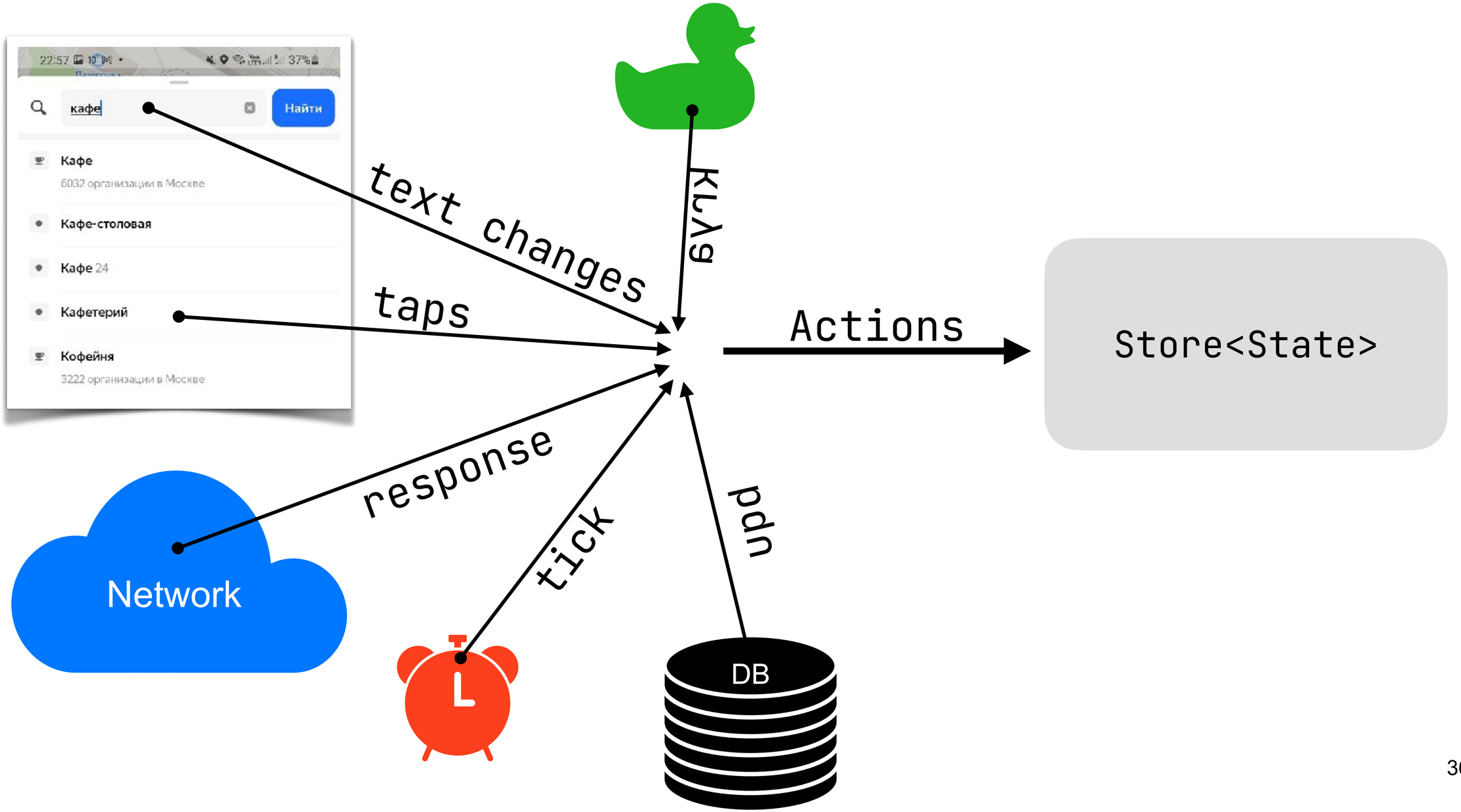


One app — multiple Stores





Actions and side-effects



Middlewares API

```
typealias Dispatch = suspend ((action: Action) → Unit)

interface Middleware<State : Any> {
    fun interfere(store: Store<State>, next: Dispatch): Dispatch
}
```

Middlewares Contract

```
typealias Dispatch = suspend ((action: Action) → Unit)

interface Middleware<State : Any> {
    fun interfere(store: Store<State>, next: Dispatch): Dispatch
}
```

Contract:

Returned Dispatch must call **next** exactly once

Middleware

```
class Store<State : Any>(
    middlewares: List<Middleware<State>>,
    initialState: State,
    private val reduce: (State, Action) → State,
) {
    …
    init {
        actionsSubject.onEach { action →
            statesSubject.emit(reduce(currentState, action))
        }.launchIn(scope)
    }
…
```

Middleware

```
class Store<State : Any>(
    middlewares: List<Middleware<State>>, …,
) {
    …
    private suspend fun reduceAndEmit(action: Action) {
        statesSubject.emit(reduce(currentState, action))
    }
    init {
         actionsSubject.onEach { action →
             reduceAndEmit(action)
         }.launchIn(scope)
     }
…
```

Middleware

```
class Store<State : Any>(
    middlewares: List<Middleware<State>>, …,
) {
    …
    private suspend fun reduceAndEmit(action: Action) {
        statesSubject.emit(reduce(currentState, action))
    }
    init {
        val reduceEmitRef: Dispatch = ::reduceAndEmit

        val dispatch: Dispatch =
            middlewares.fold(reduceEmitRef) { next, middleware →
                middleware.interfere(this, next)
            }

        actionsSubject.onEach(dispatch)
            .launchIn(scope)
    }
```

Dispatch magic

```
actionsSubject.onEach {action→
  dispatch@Middleware1 {action→
    dispatch@Middleware2 {action→
        …
            dispatch@MiddlewareN {action→
                emit(reduce(state, action))
            }
    }
}
```

Example: Analytics Middleware

```
interface AnalyticsDelegate<State> {
  fun beforeStateChanged(action: Action, oldState: State)
  fun afterStateChanged(action: Action, oldState: State, newState: State)
}
```

Example: Analytics Middleware

```
class AnalyticsMiddleware<State : Any>(
    private val delegate: AnalyticsDelegate<State>,
) : Middleware<State> {

    override fun interfere(store: Store<State>, next: Dispatch): Dispatch {
        return dispatch@{ action →
            val oldState = store.currentState
            delegate.beforeStateChanged(action, oldState)
            next(action)
            delegate.afterStateChanged(
                action, oldState = oldState, newState = store.currentState
            )
        }
    }
}
```

Example: Analytics Middleware

```
class AnalyticsMiddleware<State : Any>(
    private val delegate: AnalyticsDelegate<State>,
) : Middleware<State> {

    override fun interfere(store: Store<State>, next: Dispatch): Dispatch {
        return dispatch@{ action →
            val oldState = store.currentState
            delegate.beforeStateChanged(action, oldState)
            next(action)
            delegate.afterStateChanged(
                action, oldState = oldState, newState = store.currentState
            )
        }
    }
}
```

Example: Analytics Middleware

```
class AnalyticsMiddleware<State : Any>(
    private val delegate: AnalyticsDelegate<State>,
) : Middleware<State> {

    override fun interfere(store: Store<State>, next: Dispatch): Dispatch {
        return dispatch@{ action →
            val oldState = store.currentState
            delegate.beforeStateChanged(action, oldState)
            next(action)
            delegate.afterStateChanged(
                action, oldState = oldState, newState = store.currentState
            )
        }
    }
}
```

Example: Analytics Middleware

```
class AnalyticsMiddleware<State : Any>(
    private val delegate: AnalyticsDelegate<State>,
) : Middleware<State> {

    override fun interfere(store: Store<State>, next: Dispatch): Dispatch {
        return dispatch@{ action →
            val oldState = store.currentState
            delegate.beforeStateChanged(action, oldState)
            next(action)
            delegate.afterStateChanged(
                action, oldState = oldState, newState = store.currentState
            )
        }
    }
}
```

Example: Analytics Middleware

```
class AnalyticsMiddleware<State : Any>(
    private val delegate: AnalyticsDelegate<State>,
) : Middleware<State> {

    override fun interfere(store: Store<State>, next: Dispatch): Dispatch {
        return dispatch@{ action →
            val oldState = store.currentState
            delegate.beforeStateChanged(action, oldState)
            next(action)
            delegate.afterStateChanged(
                action, oldState = oldState, newState = store.currentState
            )
        }
    }
}
```

Dispatch magic

```
class AnalyticsMiddleware<State : Any>(
    private val delegate: AnalyticsDelegate<State>,
) : Middleware<State> {

    override fun interfere(store:
        return dispatch@{ action
            val oldState = store.
            delegate.beforeStateC
            next(action)
            delegate.afterStateCh
                action, oldState
            )
        }
    }
}
```

```
dispatch@MiddlewareX {action→
    dispatch@MiddlewareY {action→
        …
         dispatch@MiddlewareN {action→
            emit(reduce(state, action))
         }
    }
}
```

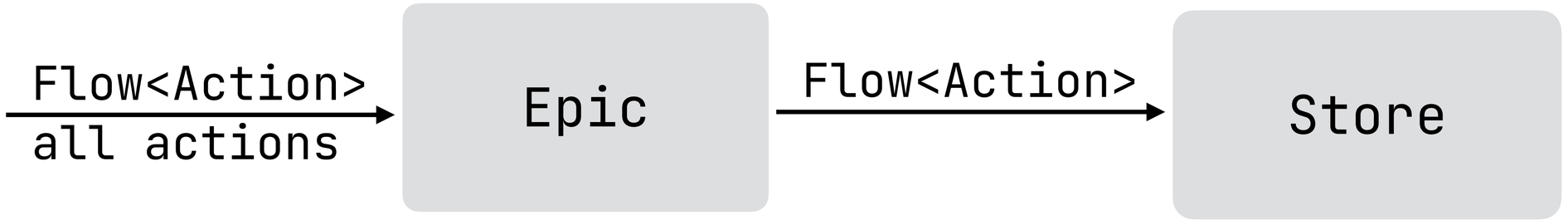
Dispatch magic

```
actionsSubject.onEach {action→
  dispatch@Middleware1 {action→
    dispatch@Middleware2 {action→
       …
        dispatch@AnalyticsMiddleware {action→
            dispatch@MiddlewareX {action→
                dispatch@MiddlewareY {action→
                 …
                    dispatch@MiddlewareN {action→
                        emit(reduce(state, action))
                    }
                }
            }
        }
    }
}
```

Example: Analytics Middleware

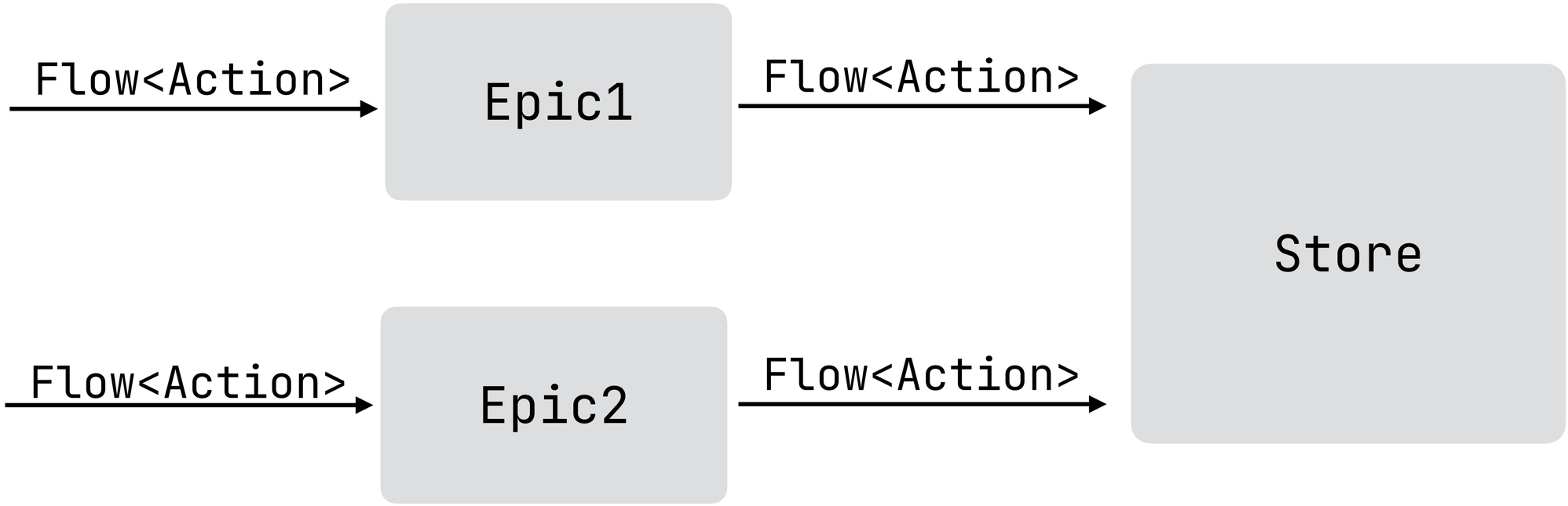
```
class AnalyticsMiddleware<State : Any>(
    private val delegate: AnalyticsDelegate<State>,
) : Middleware<State> {

    override fun interfere(store: Store<State>, next: Dispatch): Dispatch {
        return dispatch@{ action →
            val oldState = store.currentState
            delegate.beforeStateChanged(action, oldState)
            next(action)
            delegate.afterStateChanged(
                action, oldState = oldState, newState = store.currentState
            )
        }
    }
}
```

Epic Middleware

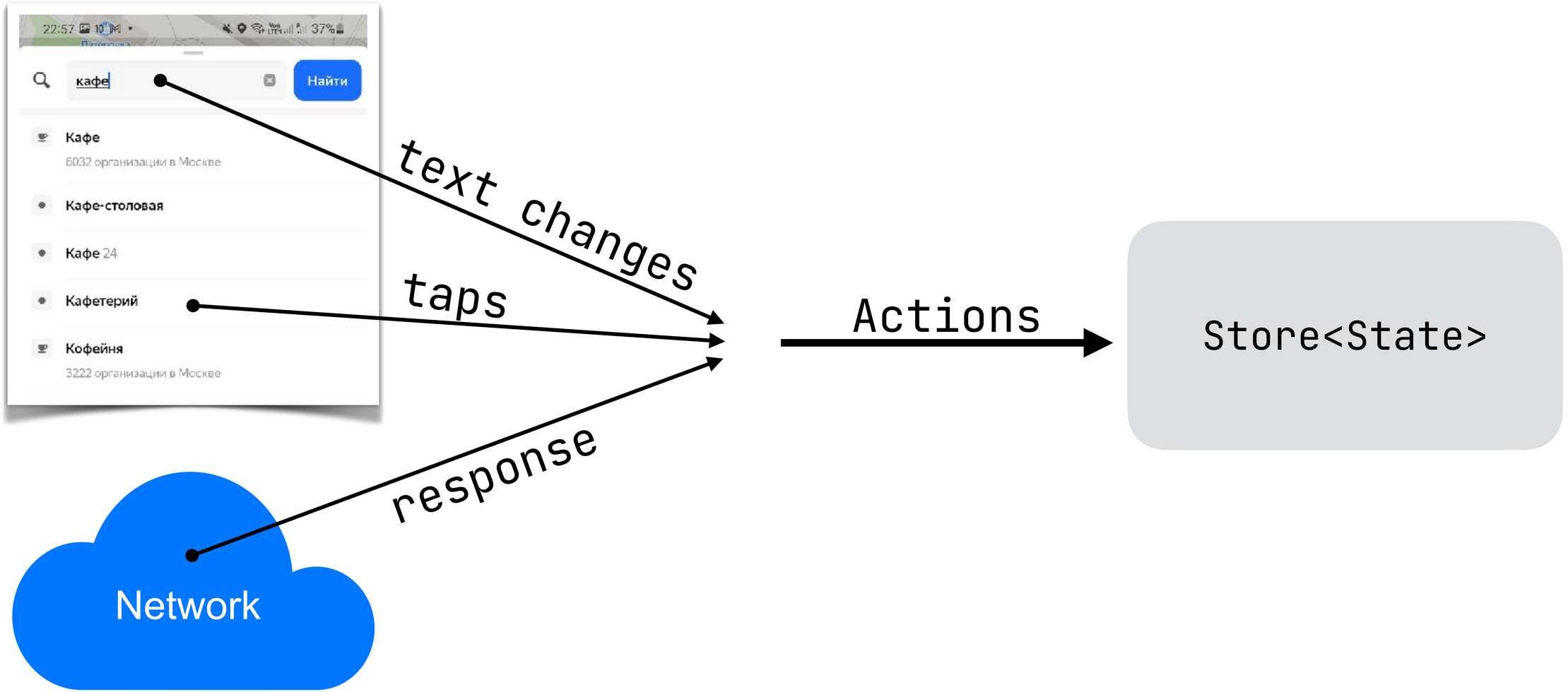



```
abstract class Epic {
    abstract fun act(actions: Flow<Action>): Flow<Action>
}
```

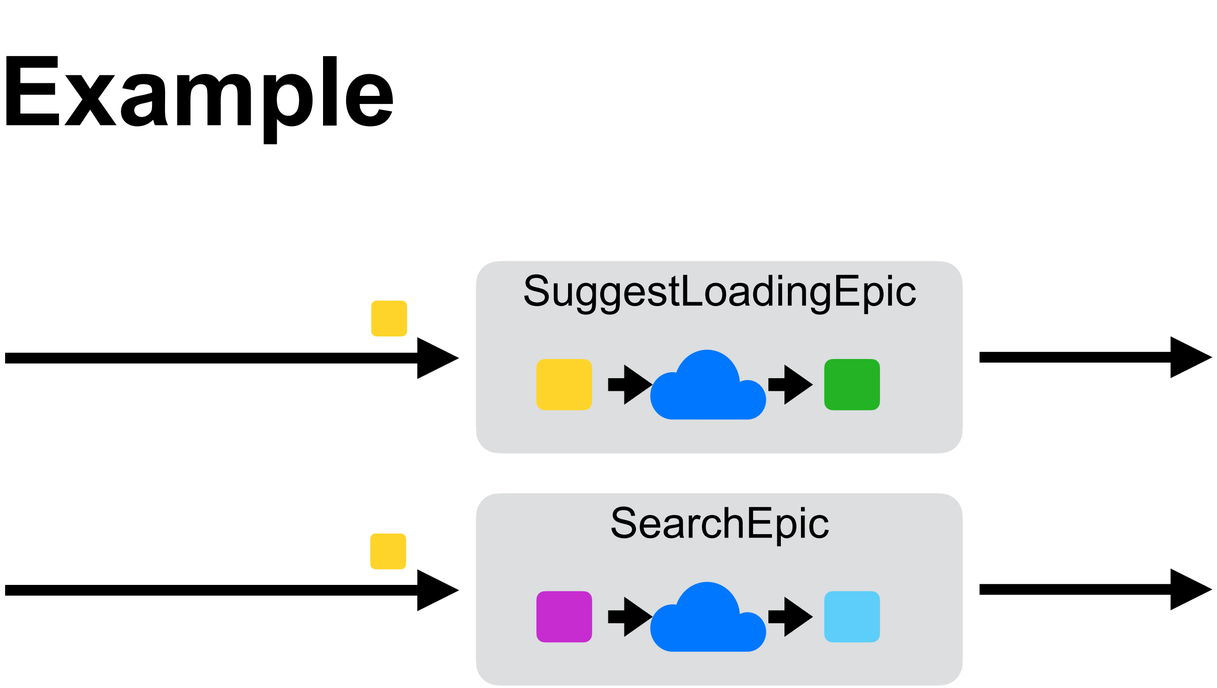
Epic Middleware



Epic Example



Example

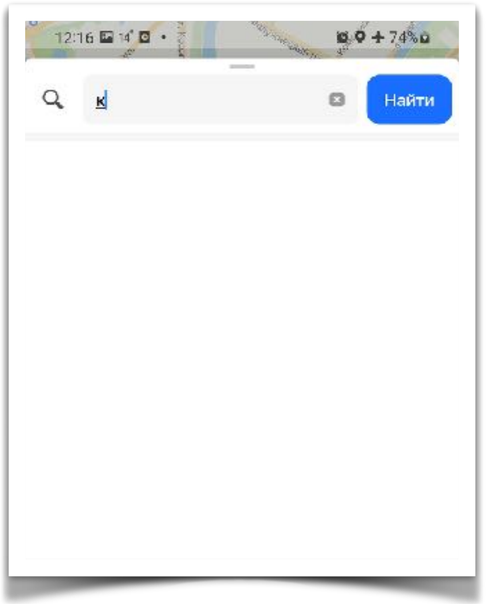


text changes

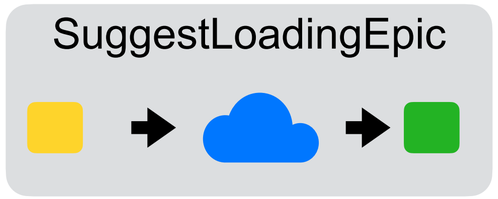
suggest loaded

suggest tap

search



Epic source example




```
class ChangeText(text): Action
class ChangeSuggest(suggest): Action
```

```
class SuggestLoadingEpic: Epic() {
    override fun act(actions: Flow<Action>): Flow<Action> {
        return actions.ofType<ChangeText>()
            .flatMapLatest { action → loadSuggest(action.text) }
            .map { list → ChangeSuggest(list) }
    }
}
```

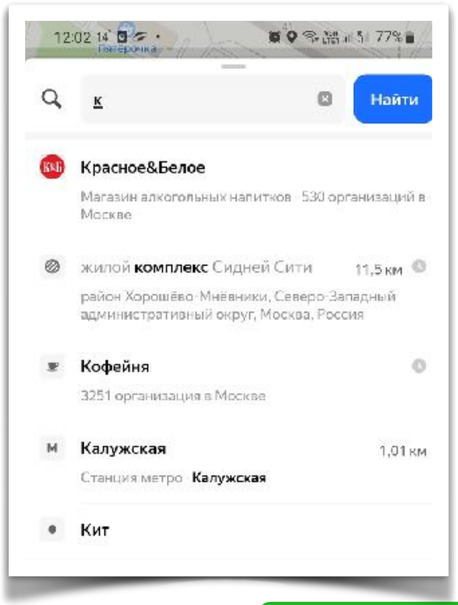
Example

text changes

suggest loaded

suggest tap

search

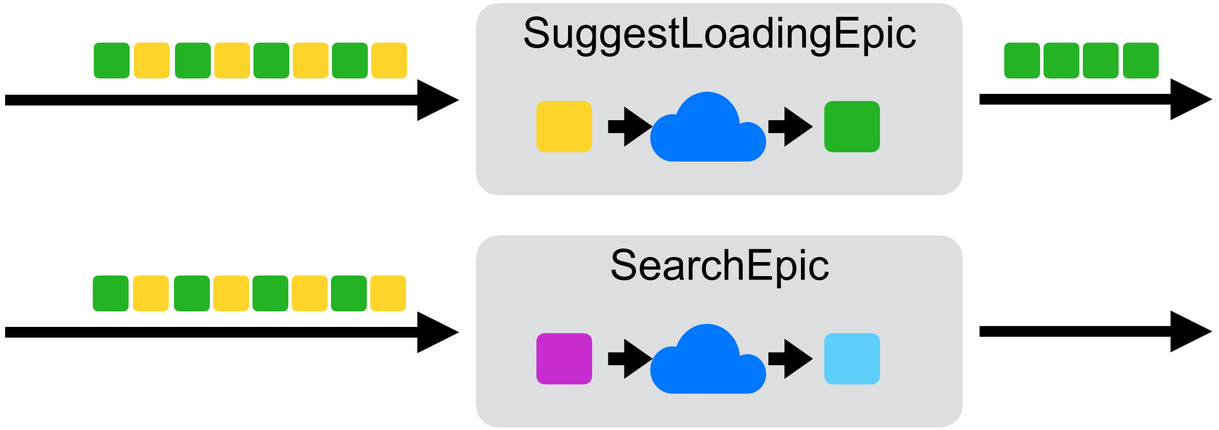


list<к>

к

Actions

Example

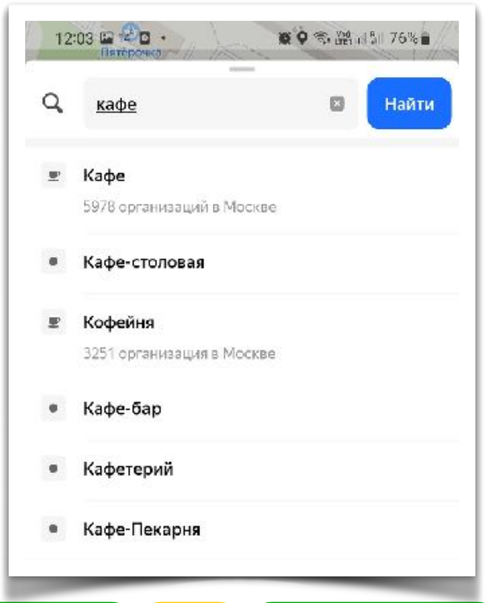


text changes

suggest loaded

suggest tap

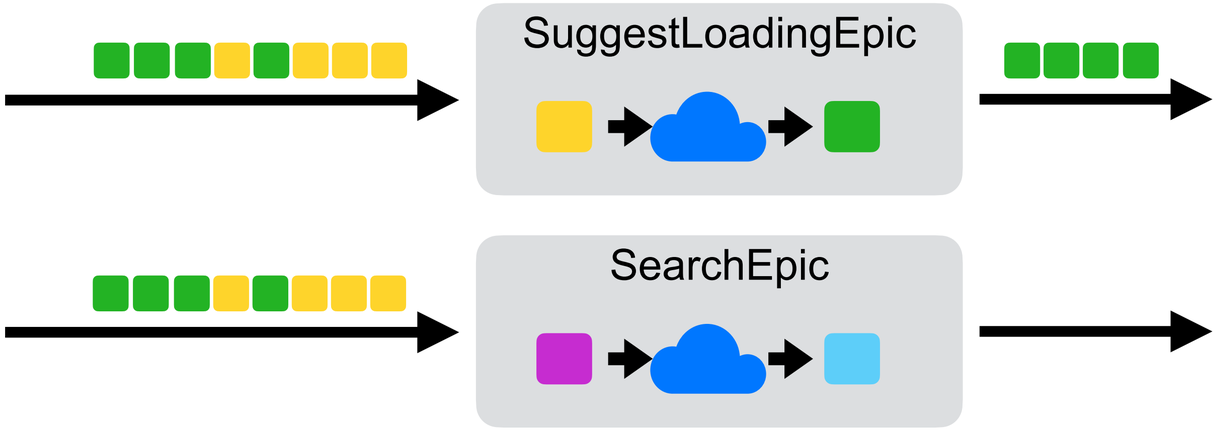
search



list<кафе> кафе list<каф> каф list<ка> ка list<к> к

Actions

Example

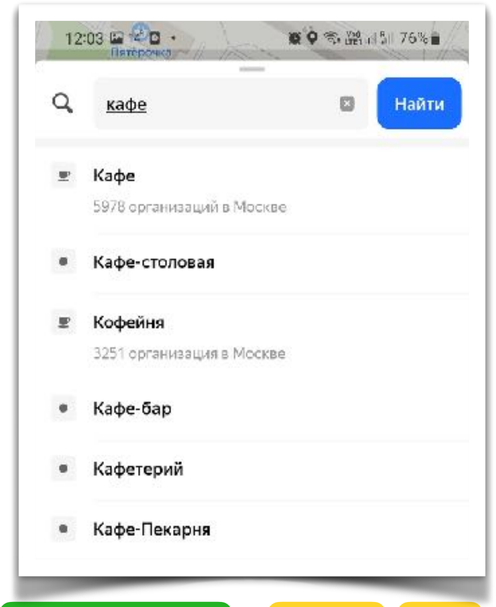


text changes

suggest loaded

suggest tap

search



list<кафе> list<каф> list<ка> кафе list<к> каф ка к

Actions

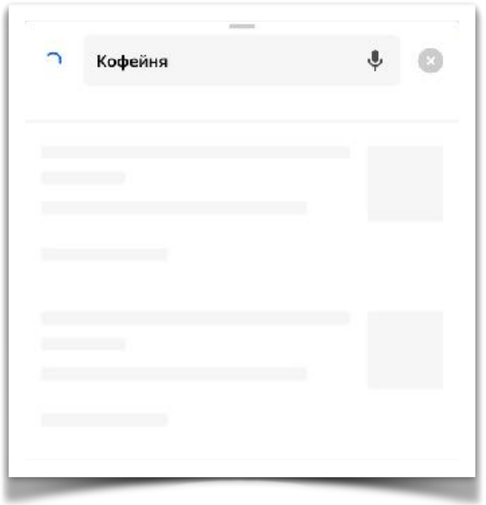
Example

text changes

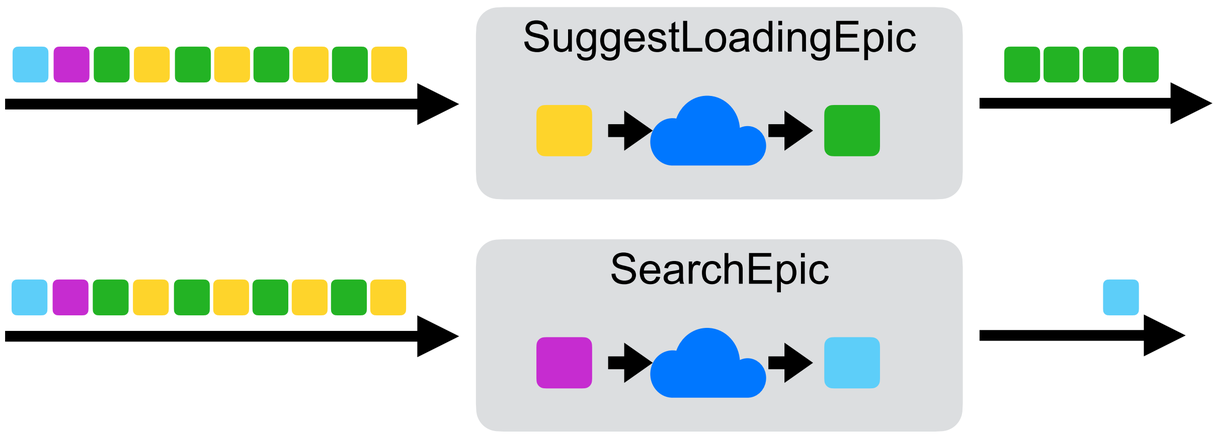
suggest loaded

suggest tap

search



Example

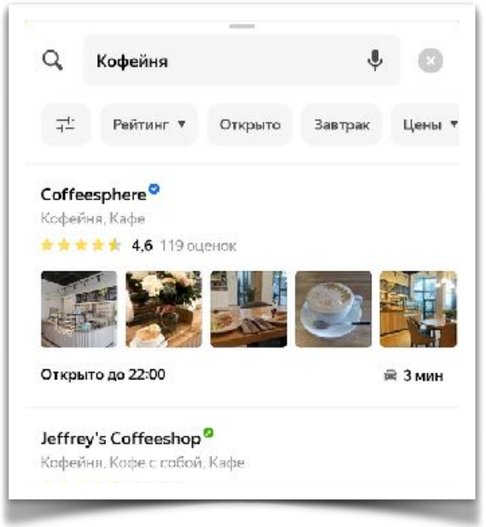


text changes

suggest loaded

suggest tap

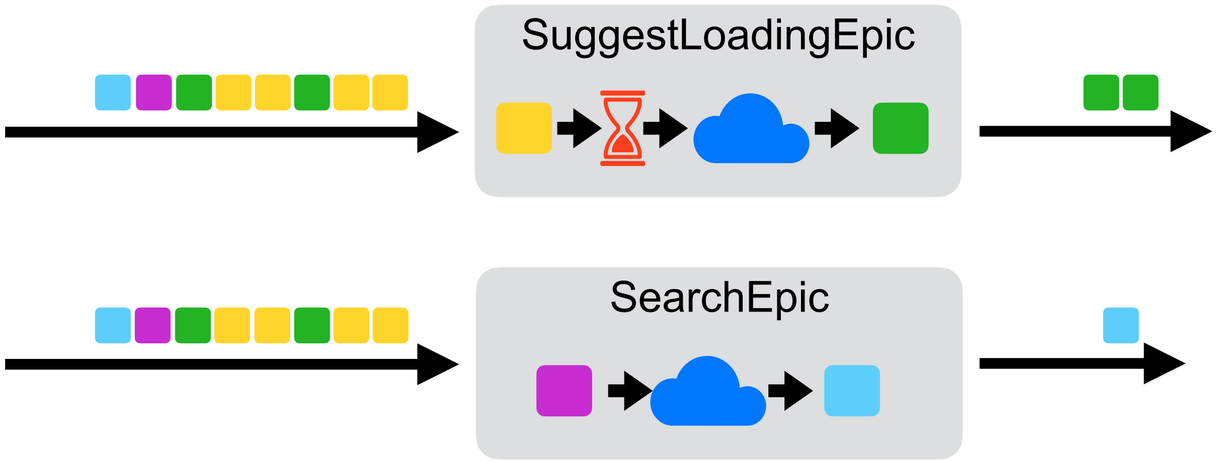
search



кофейня | кофейня | list<кафе> | кафе | list<каф> | каф | list<ка> | ка | list<к> | к

Actions

Logic in Epic

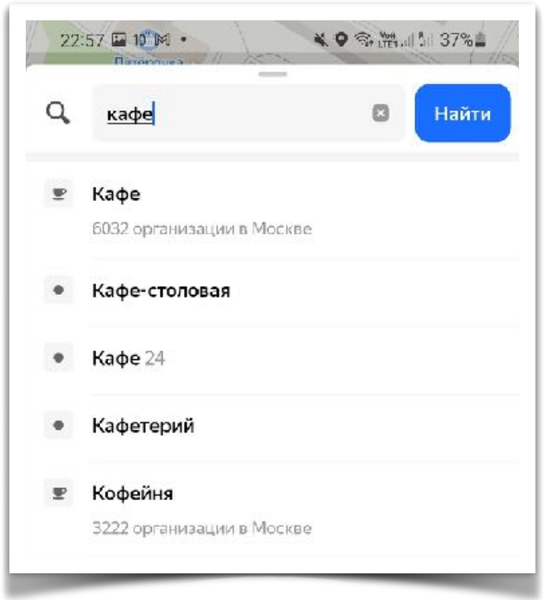


text changes

suggest loaded

suggest tap

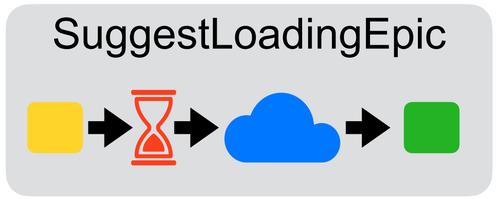
search



кофейня | кофейня | list<кафе> | кафе | каф | list<ка> | ка | к

Actions

Epic source example




```
class ChangeText(text): Action
class ChangeSuggest(suggest): Action
```

```
class SuggestLoadingEpic: Epic() {
    override fun act(actions: Flow<Action>): Flow<Action> {
        return actions.ofType<ChangeText>()
            .debounce(300L)
            .flatMapLatest { action → loadSuggest(action.text) }
            .map { list → ChangeSuggest(list) }
    }
}
```

Redux: positive and negative

Redux: positive

- Logic decomposition
- Easy testing
- Multiplatform state management
- Single source of truth

Redux: positive

- **Logic decomposition**
- Easy testing
- Multiplatform state management
- Single source of truth

> `Pure fun reduce()`

> `Multiple single-responsible Epics`

Redux: positive

- Logic decomposition
- **Easy testing**
- Multiplatform state management
- Single source of truth

> `Pure fun reduce() **again!**`

> `Multiple single-responsible Epics **again!**`

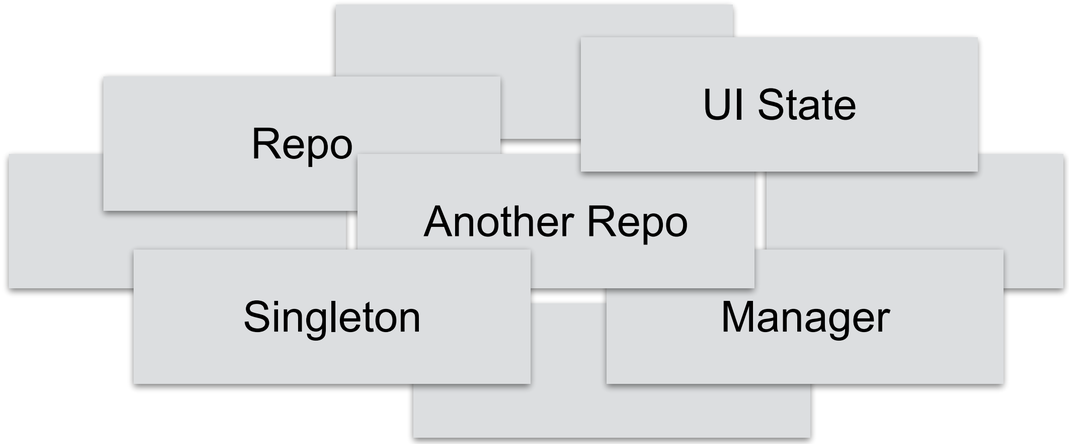
Redux: positive

- Logic decomposition
- Easy testing
- **Multiplatform (immutable!) state management**
- Single source of truth

Redux: positive

- Logic decomposition
- Easy testing
- Multiplatform state management
- **Single source of truth**

```
Store<State>
```

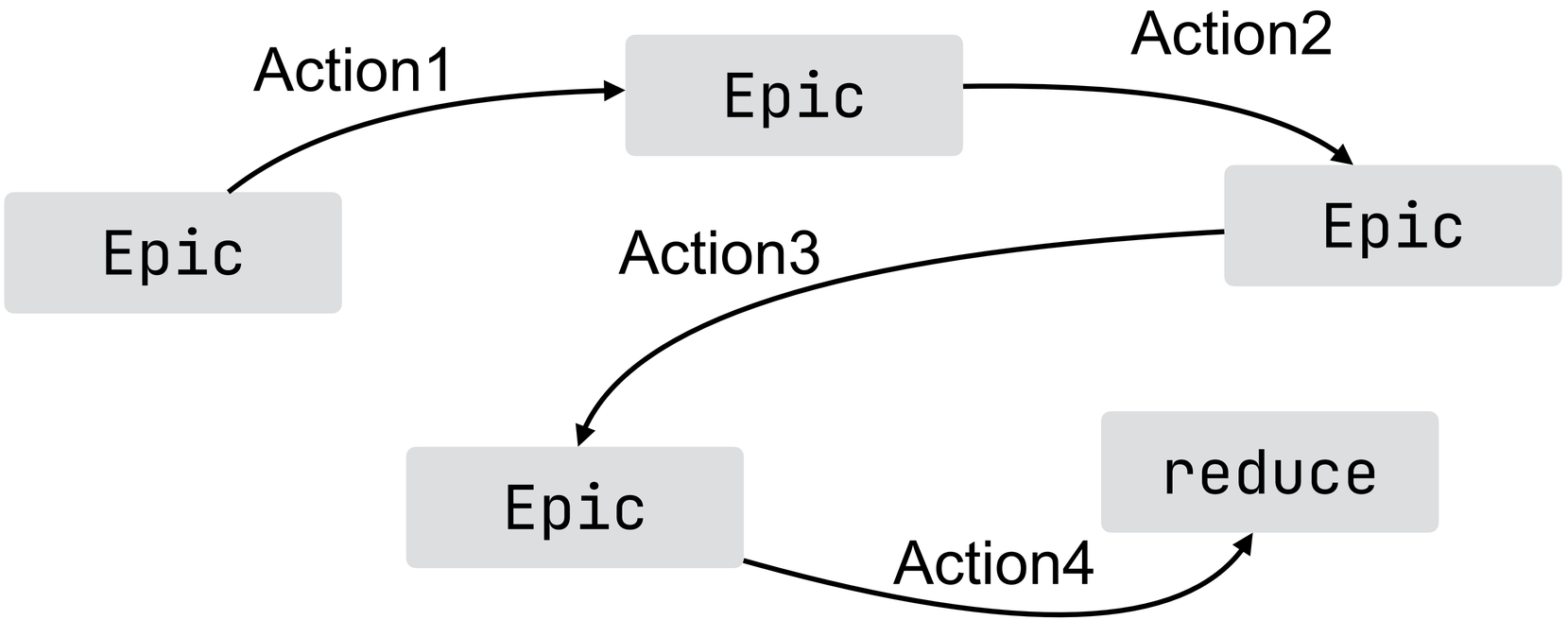


Redux: negative

- Implicit control flow
- No commands to view
- Large business logic API
- Different state interpretation on iOS and Android

Redux: negative

- **Implicit control flow**
- No commands to view
- Large business logic API
- Different state interpretation on iOS and Android

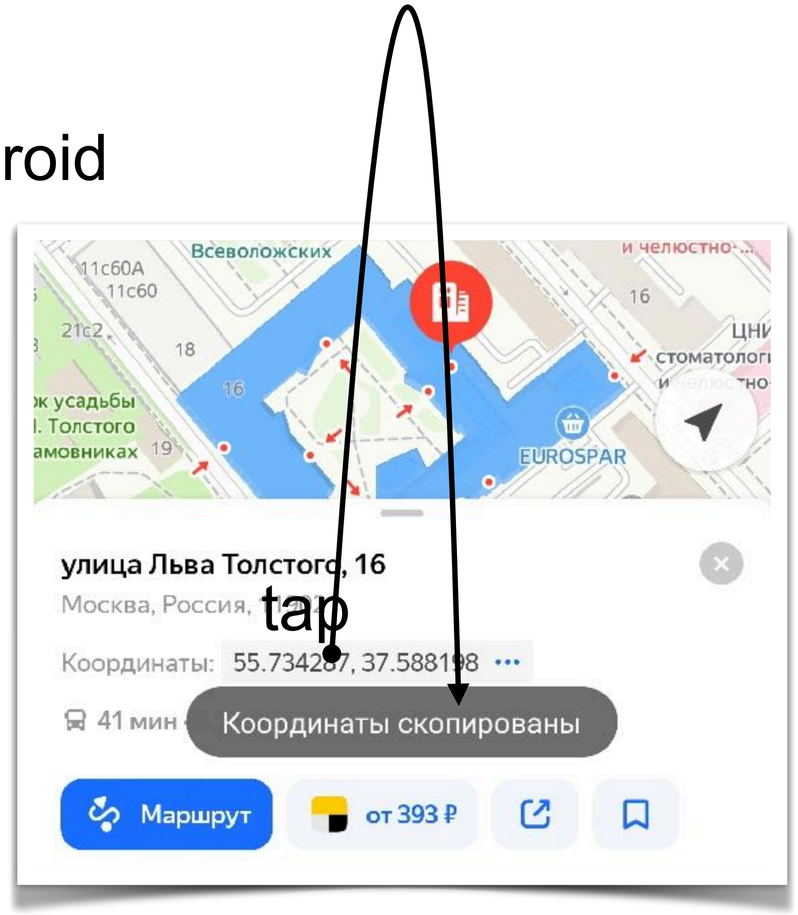


Redux: negative

- Implicit control flow
- **No commands to view**
- Large business logic API
- Different state interpretation on iOS and Android

```
Fire and forget
```

```
Scrolls
```



Redux: negative

- Implicit control flow
- No commands to view
- **Large business logic API**
- Different state interpretation on iOS and Android

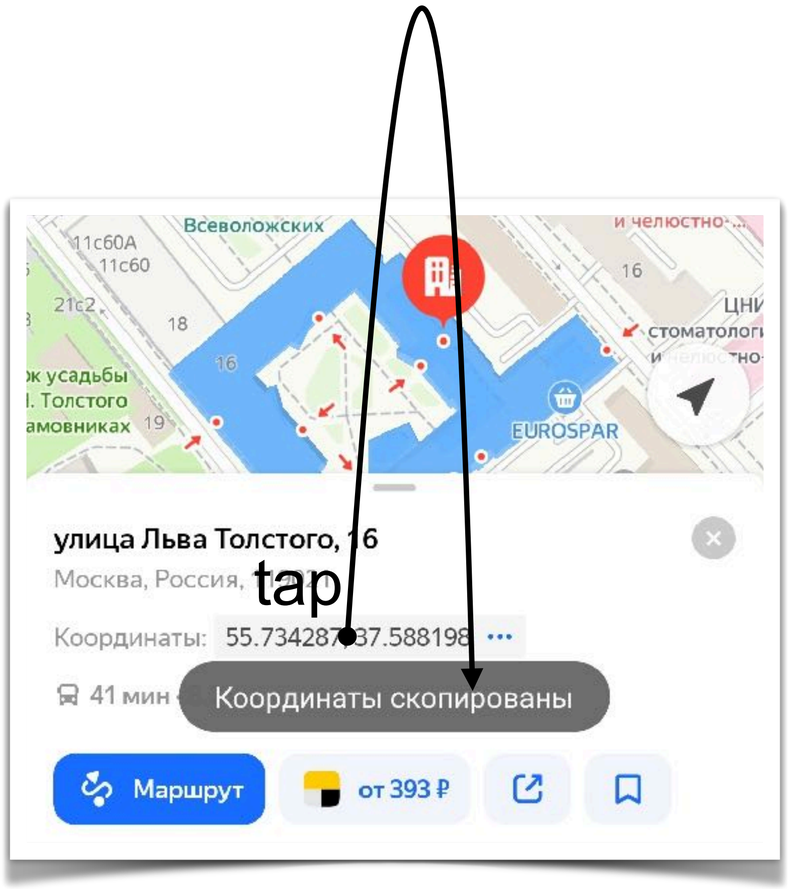
> Every Action can be treated as public Method

Redux: negative

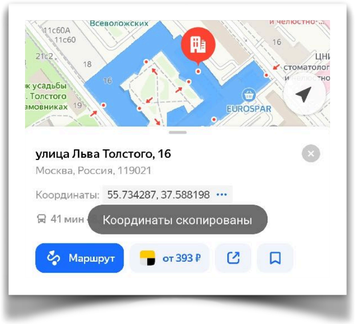
- Implicit control flow
- No commands to view
- Large business logic API
- **Different state interpretation on iOS and Android**

No commands to Views problem

Example: Toast (Fire and forget)

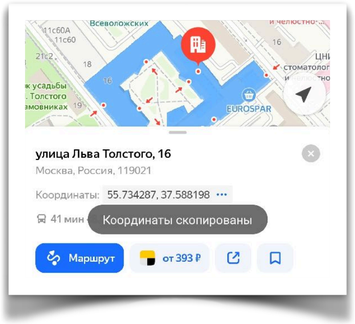


Hmmm… Make it in UI?




```
class CardFragment: Fragment() {
    @Inject lateinit var store: Store<CardState>
    @Inject lateinit var clipboard: Clipboard

    override fun onViewCreated(…) {
        coordinatesButton.onClick {
            clipboard.copy(store.currentState.coordinates)
            Toast.makeText(…).show()
        }
    }
}
```

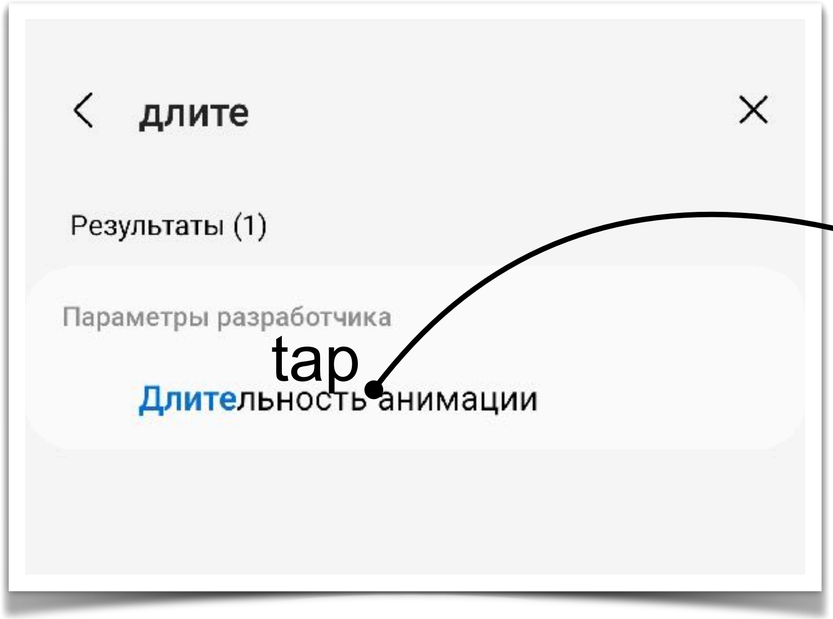
Hmmm… Make it on platform? Nooo!

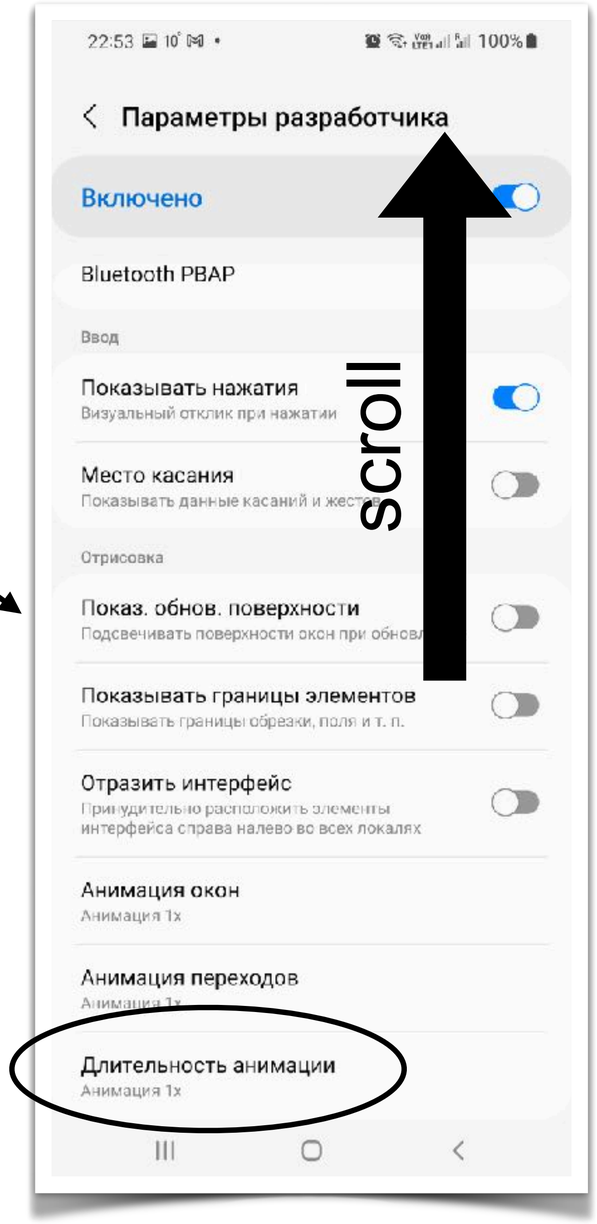



```
class CardFragment: Fragment() {
    @Inject lateinit var store: Store<CardState>
    @Inject lateinit var clipboard: Clipboard

    override fun onViewCreated(…) {
        coordinatesButton.onClick {
            clipboard.copy(store.currentState.coordinates)
            Toast.makeText(…).show()
        }
    }
}
```

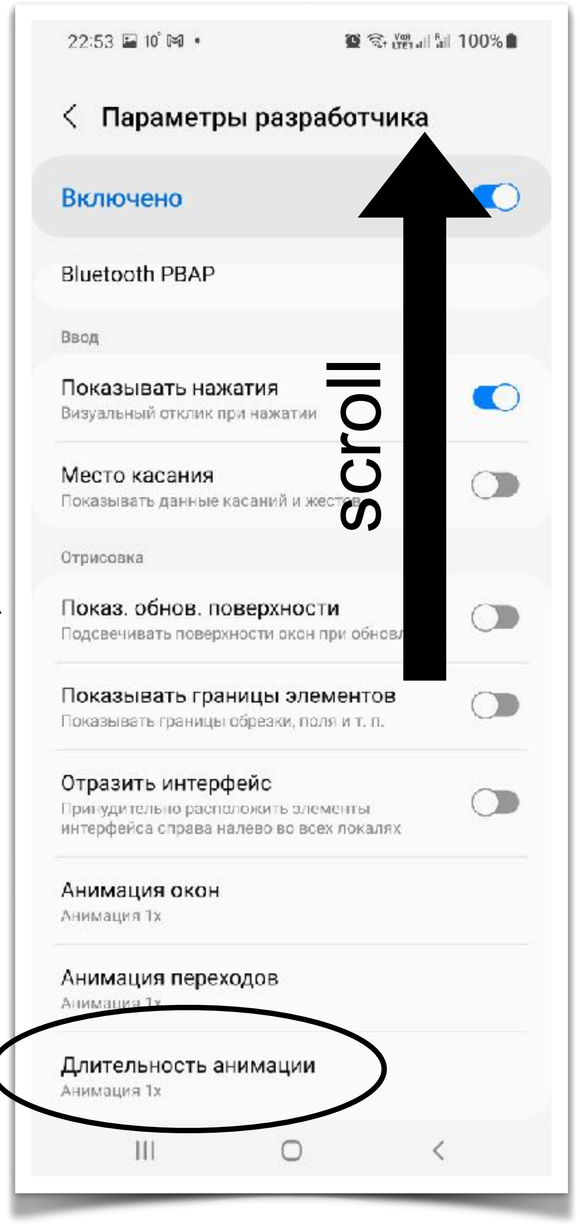
NO MULTIPLATFROM

Example: Scroll





Example: Scroll



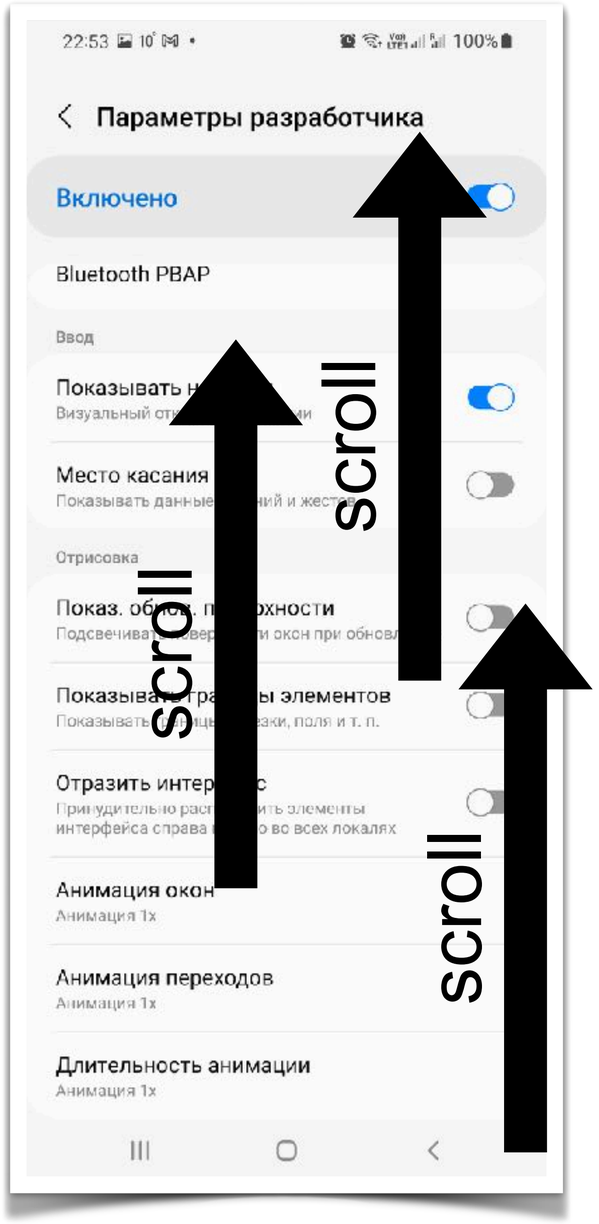
`Store<State>` → `States`

Example: Scroll

```
data class State(
    val scrollPosition: Int,
)
```

Store<State>

S1, S2, S3, …



View as Side effect

View as Side effect

```
interface Scroller {
    fun scrollTo(position: Int)
}

class ScrollAction(val position: Int): Action

class ScrollEpic @Inject constructor(
    val scroller: Scroller,
): Epic() {
    override fun act(actions: Flow<Action>): Flow<Action> {
        return actions
            .ofType<ScrollAction>()
            .onEach { scroller.scrollTo(position = it.position) }
            .skipAll()
    }
}
```

Side effect implemented on iOS / Android

```
interface Scroller {
    fun scrollTo(position: Int)
}

class ScrollAction(val position: Int): Action

class ScrollEpic @Inject constructor(
    val scroller: Scroller,
): Epic() {
    override fun act(actions: Flow<Action>): Flow<Action> {
        return actions
            .ofType<ScrollAction>()
            .onEach { scroller.scrollTo(position = it.position) }
            .skipAll()
    }
}
```

Action command

```
interface Scroller {
    fun scrollTo(position: Int)
}

class ScrollAction(val position: Int): Action

class ScrollEpic @Inject constructor(
    val scroller: Scroller,
): Epic() {
    override fun act(actions: Flow<Action>): Flow<Action> {
        return actions
            .ofType<ScrollAction>()
            .onEach { scroller.scrollTo(position = it.position) }
            .skipAll()
    }
}
```

Epic implementation trivial

```
interface Scroller {
    fun scrollTo(position: Int)
}

class ScrollAction(val position: Int): Action

class ScrollEpic @Inject constructor(
    val scroller: Scroller,
): Epic() {
    override fun act(actions: Flow<Action>): Flow<Action> {
        return actions
            .ofType<ScrollAction>()
            .onEach { scroller.scrollTo(position = it.position) }
            .skipAll()
    }
}
```

Toast example

```
interface Notifier {
    fun notify(text: String)
}

class NotifyAction(val text: String): Action

class NotifyEpic @Inject constructor(
    val notifier: Notifier,
): Epic() {
    override fun act(actions: Flow<Action>): Flow<Action> {
        return actions
            .ofType<NotifyAction>()
            .onEach { notifier.notify(text = it.text) }
            .skipAll()
    }
}
```

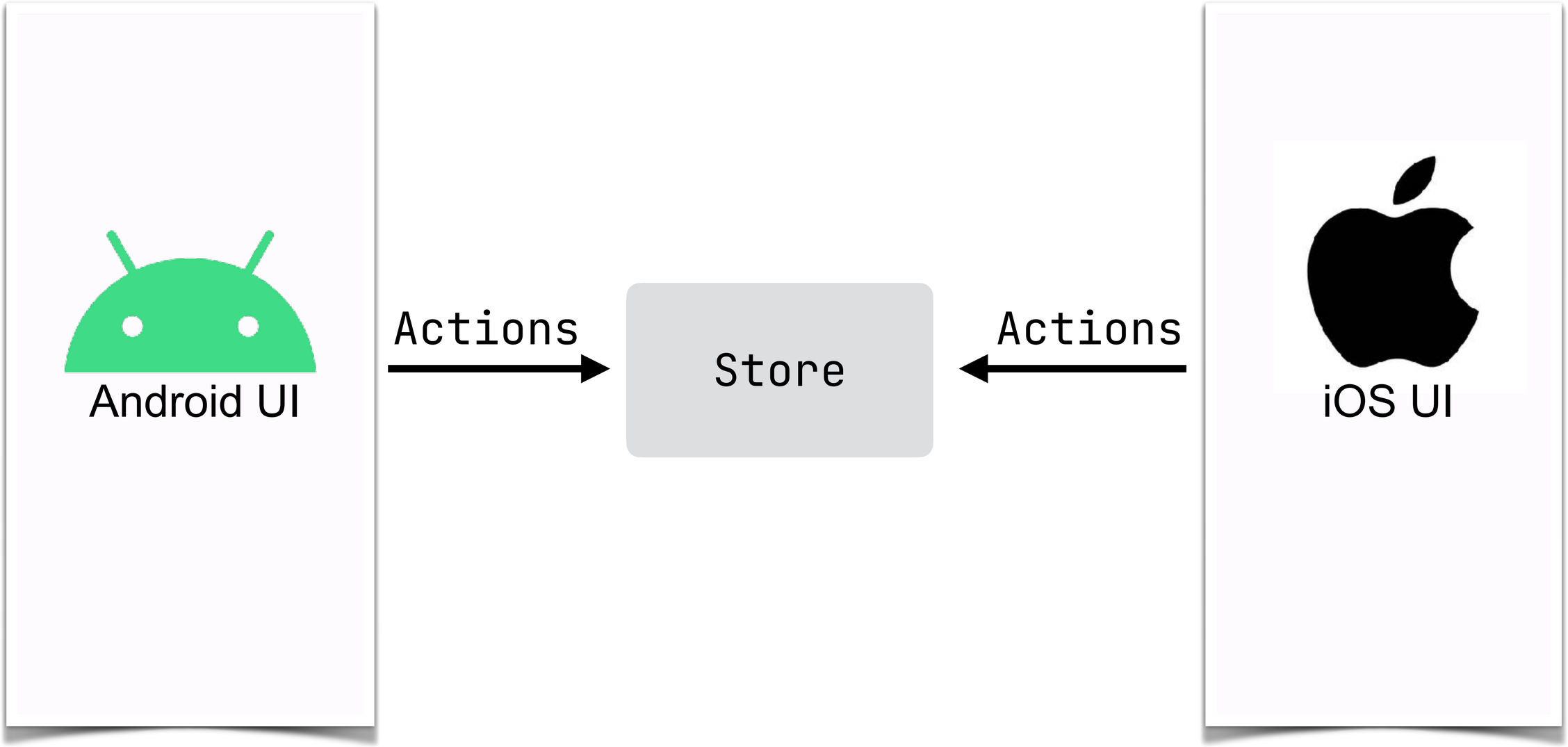
Clipboard example

```
interface Clipboard {
    fun copy(text: String)
}

class CopyToClipboardAction(val text: String): Action

class ClipboardEpic @Inject constructor(
    val clipboard: Clipboard,
): Epic() {
    override fun act(actions: Flow<Action>): Flow<Action> {
        return actions
            .ofType<CopyToClipboardAction>()
            .onEach { clipboard.copy(text = it.text) }
            .map { NotifyAction("Copied to clipboard") }
    }
}
```

Clipboard example

```
interface Clipboard {
    fun copy(text: String)
}

class CopyToClipboardAction(val text: String): Action

class ClipboardEpic @Inject constructor(
    val clipboard: Clipboard,
): Epic() {
    override fun act(actions: Flow<Action>): Flow<Action> {
        return actions
            .ofType<CopyToClipboardAction>()
            .onEach { clipboard.copy(text = it.text) }
            .map { NotifyAction("Copied to clipboard") }
    }
}
```

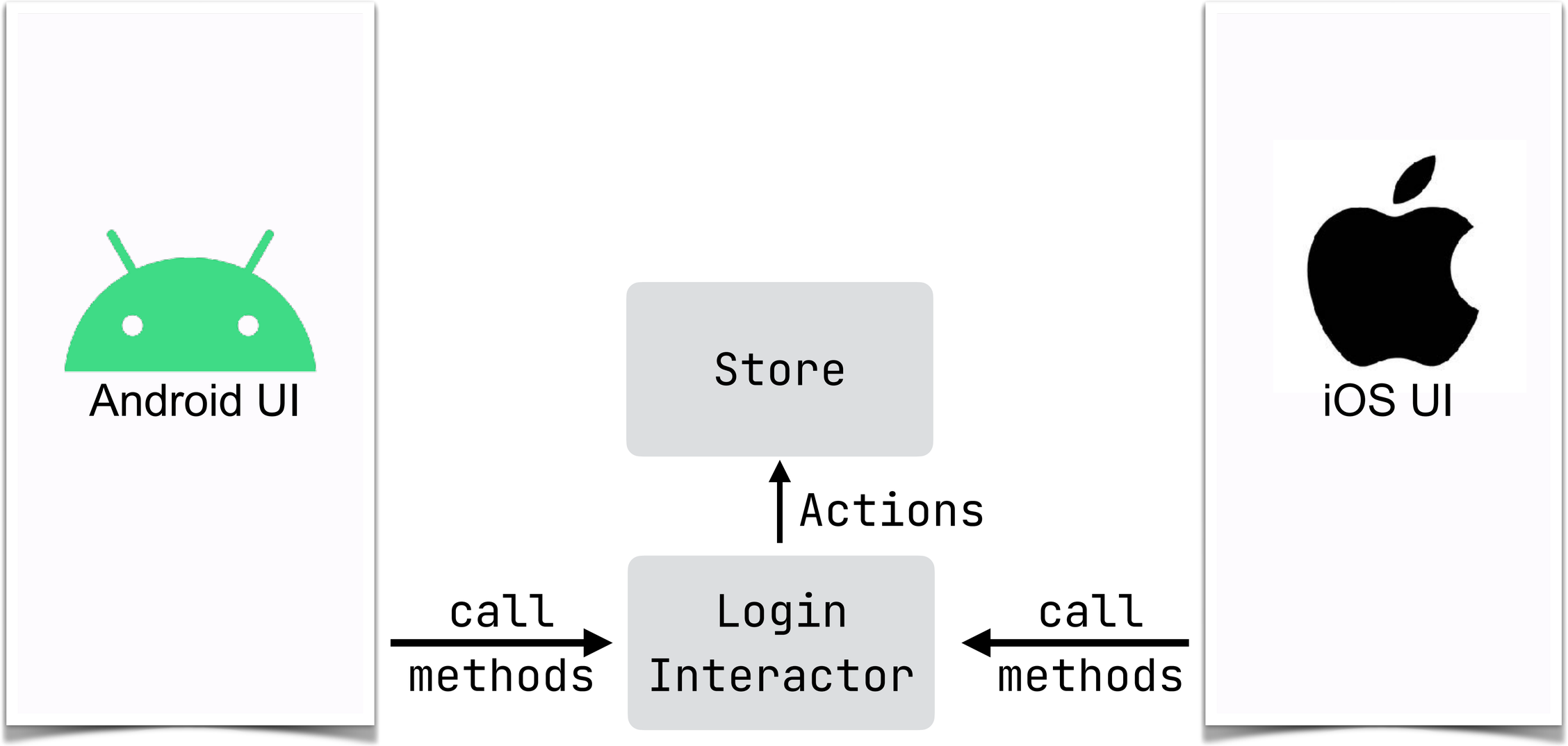
Large API problem



Reduce Large API: Interactors

```
interface LoginInteractor {
    fun loginChanged(login: String)
    fun passwordChanged(password: String)
    fun loginRequested()
}
```

Reduce Large API and boilerplate



Trivial Interactor implementation

```
class LoginInteractorImpl(
    private val store: Store<LoginState>,
) : LoginInteractor {
    override fun loginChanged(login: String) {
        store.dispatch(ChangeLogin(login))
    }

    override fun passwordChanged(password: String) {
        store.dispatch(ChangePassword(password))
    }

    override fun loginRequested() {
        store.dispatch(PerformLogin)
    }
}
```

Trivial Interactor ~~implementation~~ boilerplate

```
class LoginInteractorImpl(
    private val store: Store<LoginState>,
) : LoginInteractor {
    override fun loginChanged(login: String) {
        store.dispatch(changeLogin(login))
    }

    override fun passwordChanged(password: String) {
        store.dispatch(changePassword(password))
    }

    override fun loginRequested() {
        store.dispatch(PerformLogin)
    }
}
```

Reduce Large API: Interactor

```
interface LoginInteractor {
    fun dispatch(action: LoginPublicAction)
}
```

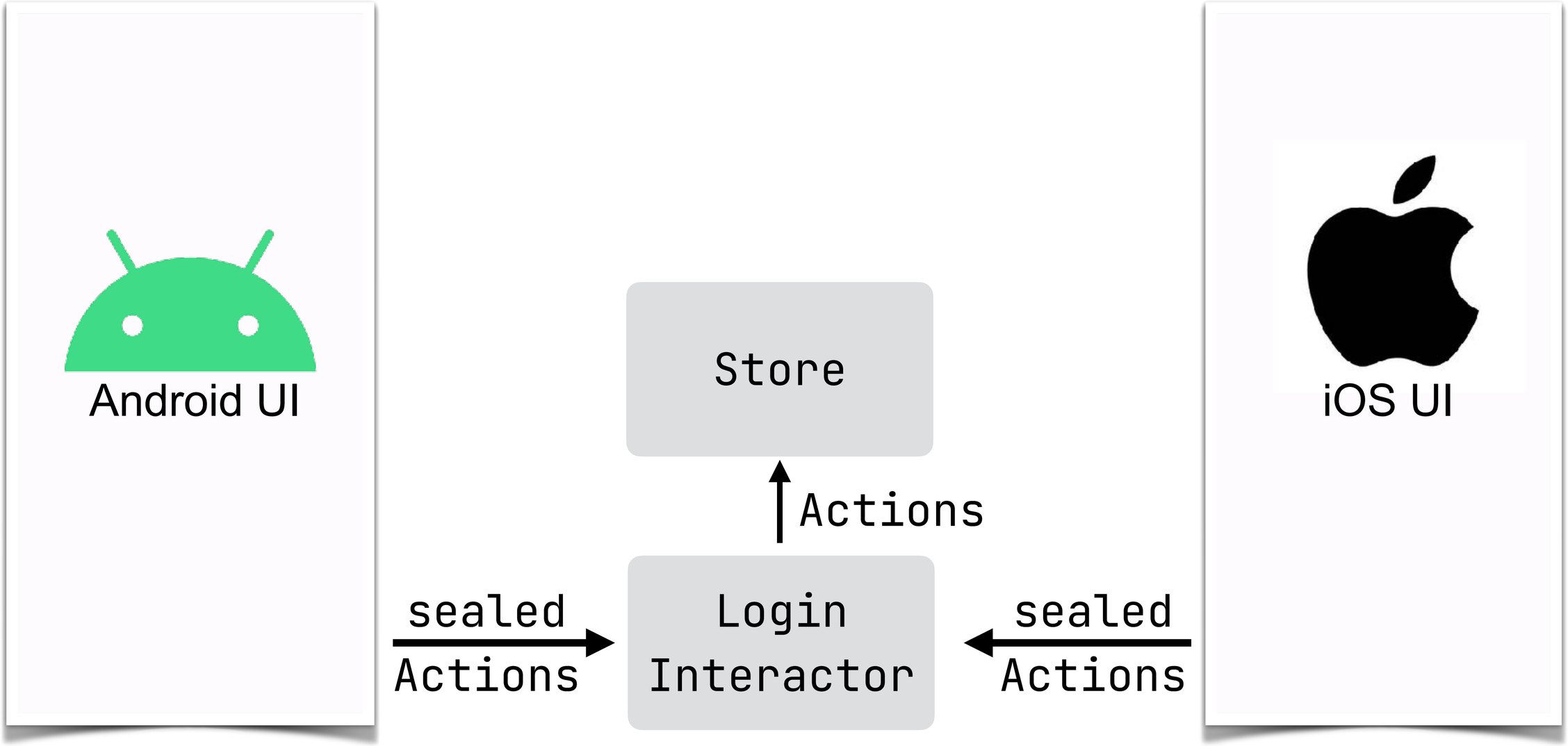
Reduce Large API: Interactor + sealed interface

```
interface LoginInteractor {
    fun dispatch(action: LoginPublicAction)
}

sealed interface LoginPublicAction: Action

class ChangeLogin(val login: String): LoginPublicAction
class ChangePassword(val password: String): LoginPublicAction
object PerformLogin: LoginPublicAction
```

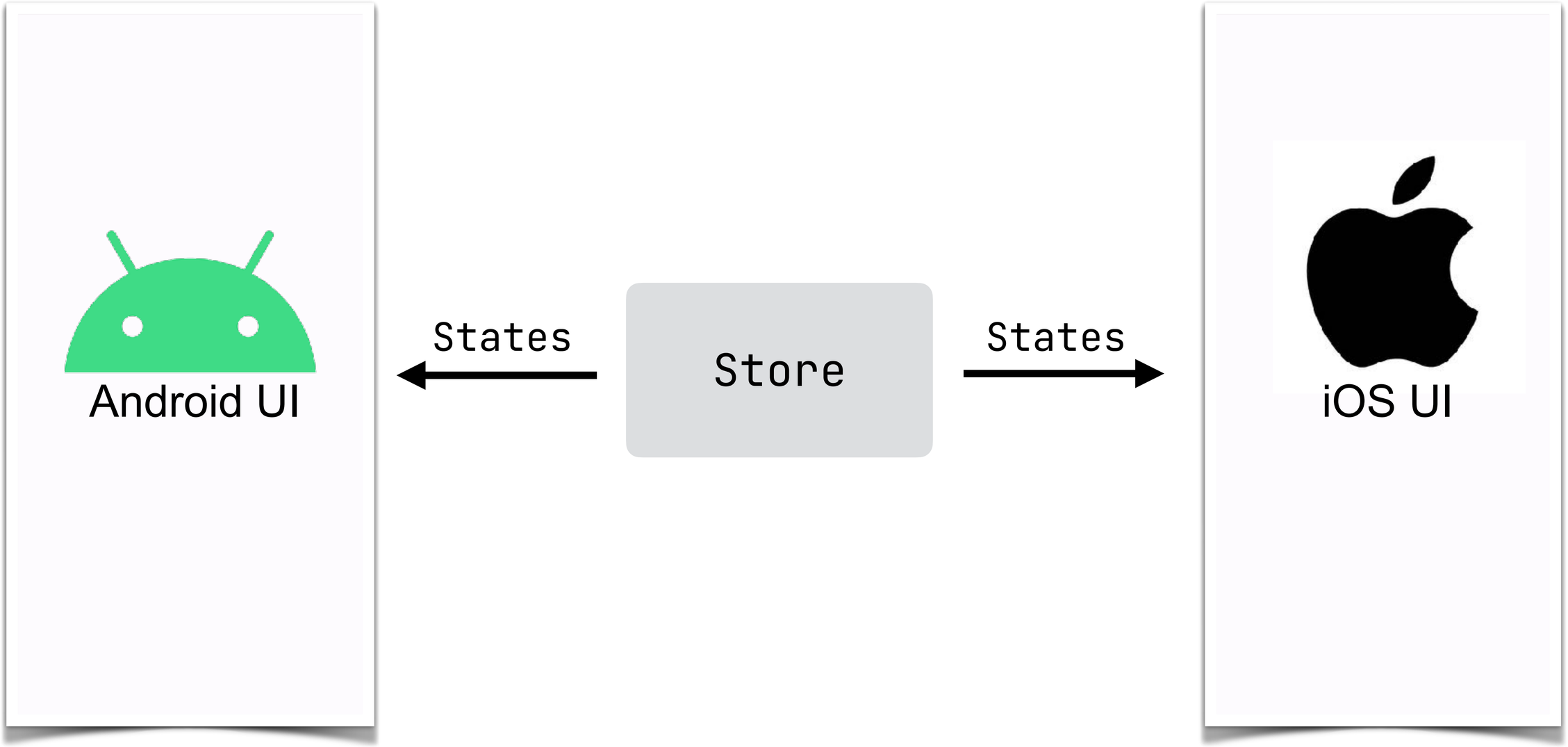
Interactor implementation trivial

```
interface LoginInteractor {
    fun dispatch(action: LoginPublicAction)
}

internal class InteractorImpl(
    private val store: Store<LoginState>,
): LoginInteractor {
    override fun dispatch(action: LoginPublicAction) {
        store.dispatch(action)
    }
}
```

Reduce Large API and boilerplate

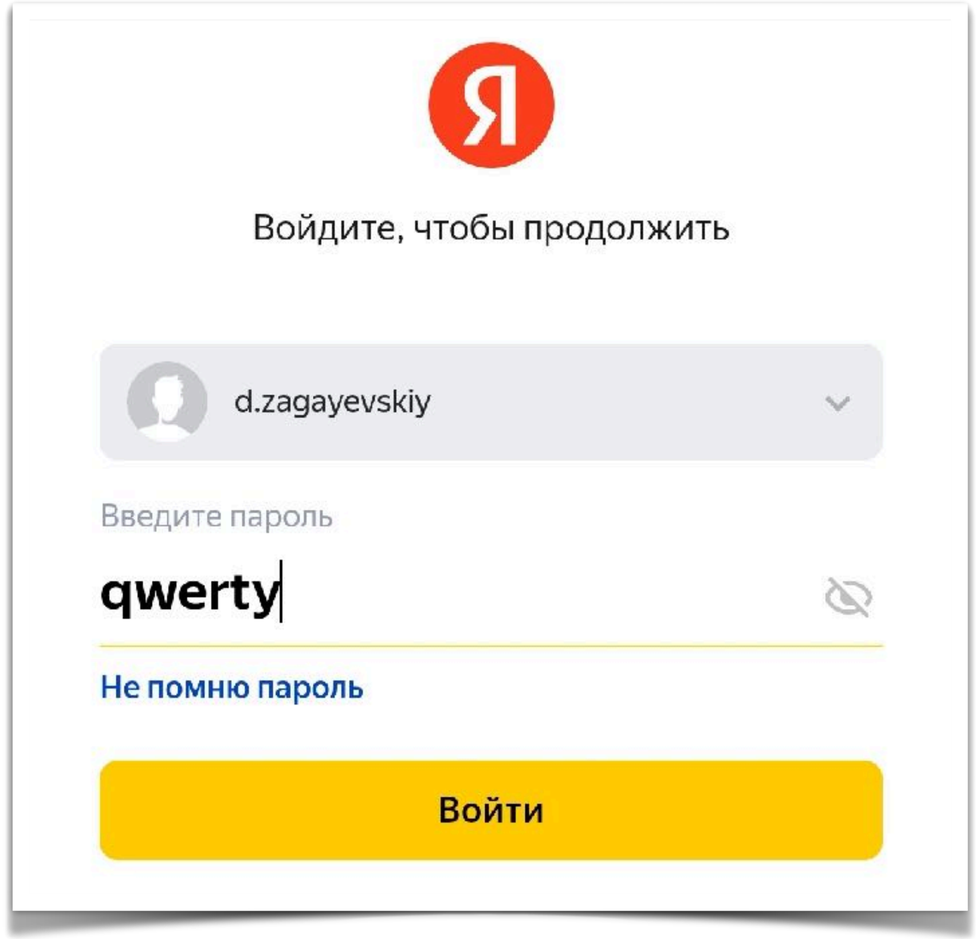


State interpretation problem



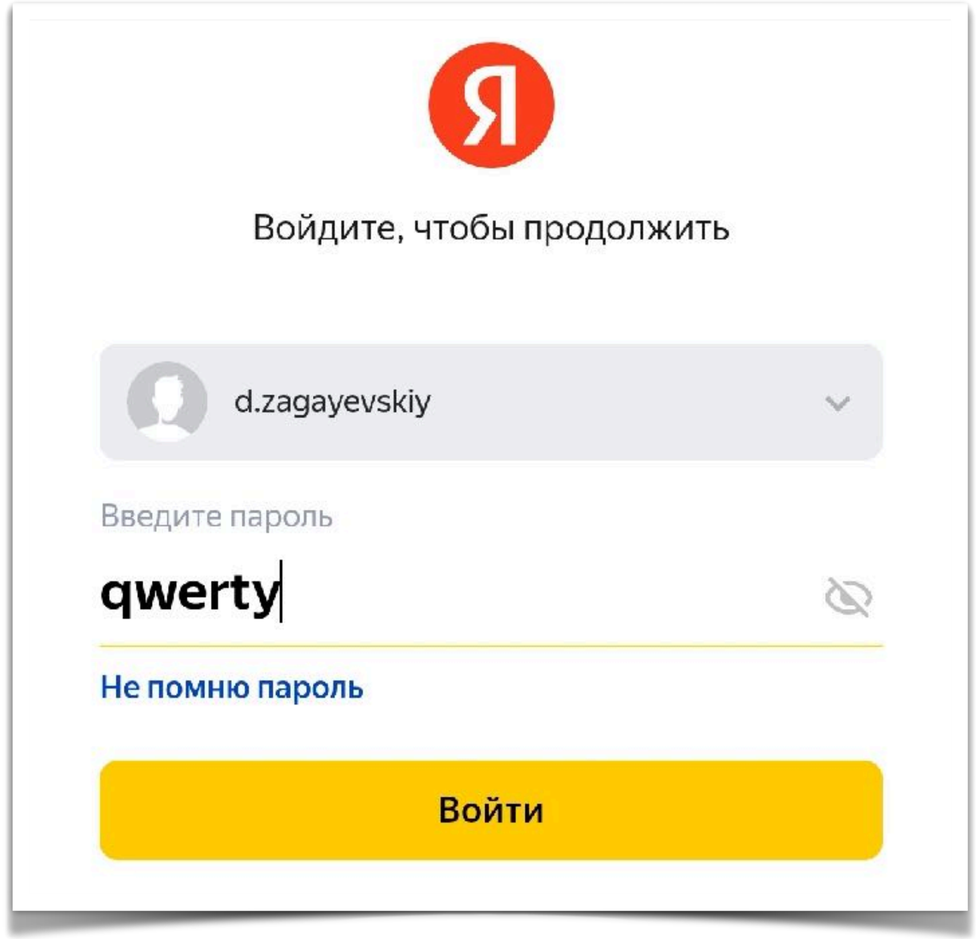
State interpretation problem

```
data class State(
    val username: String,
    val password: String?,
    val passwordHidden: Boolean,
    val imageUrl: String?,
)
```



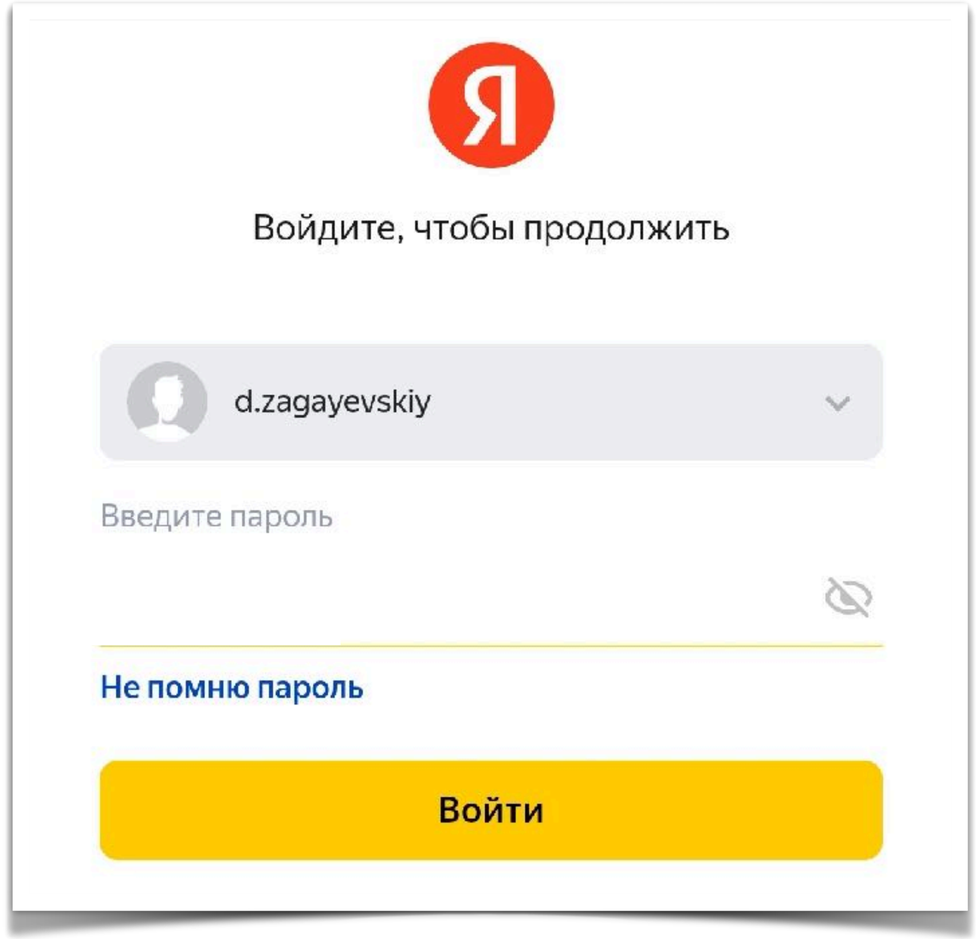
State interpretation naive solution

```
data class State(
    val username: String,
    val password: String?,
    val passwordHidden: Boolean,
    val imageUrl: String?,
    val enterActive: Boolean,
)
```



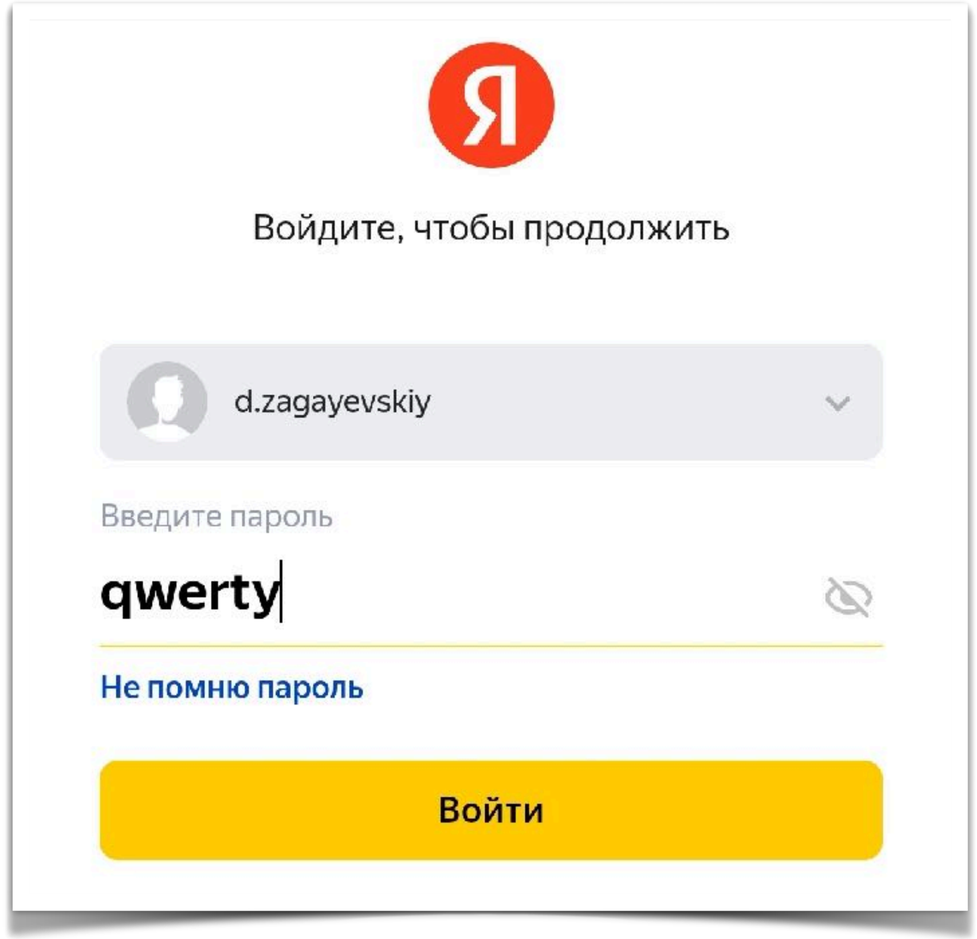
State interpretation ~~naive~~ wrong solution

```
data class State(
    val username: String,
    val password: String?,
    val passwordHidden: Boolean,
    val imageUrl: String?,
    val enterActive: Boolean,
)
state.copy(
    password = null,
    enterActive = true,
)
```



Multiplatform ViewState

```
class ViewState(
    val username: String,
    val password: String?,
    val passwordHidden: Boolean,
    val imageUrl: String?,
    val enterActive: Boolean,
)
```



Multiplatform ViewStateMapping

```
fun State.toViewState() = ViewState(
    username = username,
    password = password,
    passwordHidden = passwordHidden,
    imageUrl = imageUrl,
    enterActive = password.isNullOrEmpty().not(),
)
```

Multiplatform ViewState + ViewStateMapping

```
class LoginViewStateMapper(
    private val store: Store<LoginState>,
) {
    fun viewStates(): PlatformStream<ViewState> {
        return store
            .states()
            .distinctUntilChanged()
            .map { it.toViewState() }
            .toPlatformStream()
    }
}
```

PlatformStream?

```
class LoginViewStateMapper(
    private val store: Store<LoginState>,
) {
    fun viewStates(): PlatformStream<ViewState> {
        return store
            .states()
            .distinctUntilChanged()
            .map { it.toViewState() }
            .toPlatformStream()
    }
}
```

K/N → ObjC -> Swift interop problem

```
interface Flow<T> → @protocol Flow ⇒ Generic erasure
```

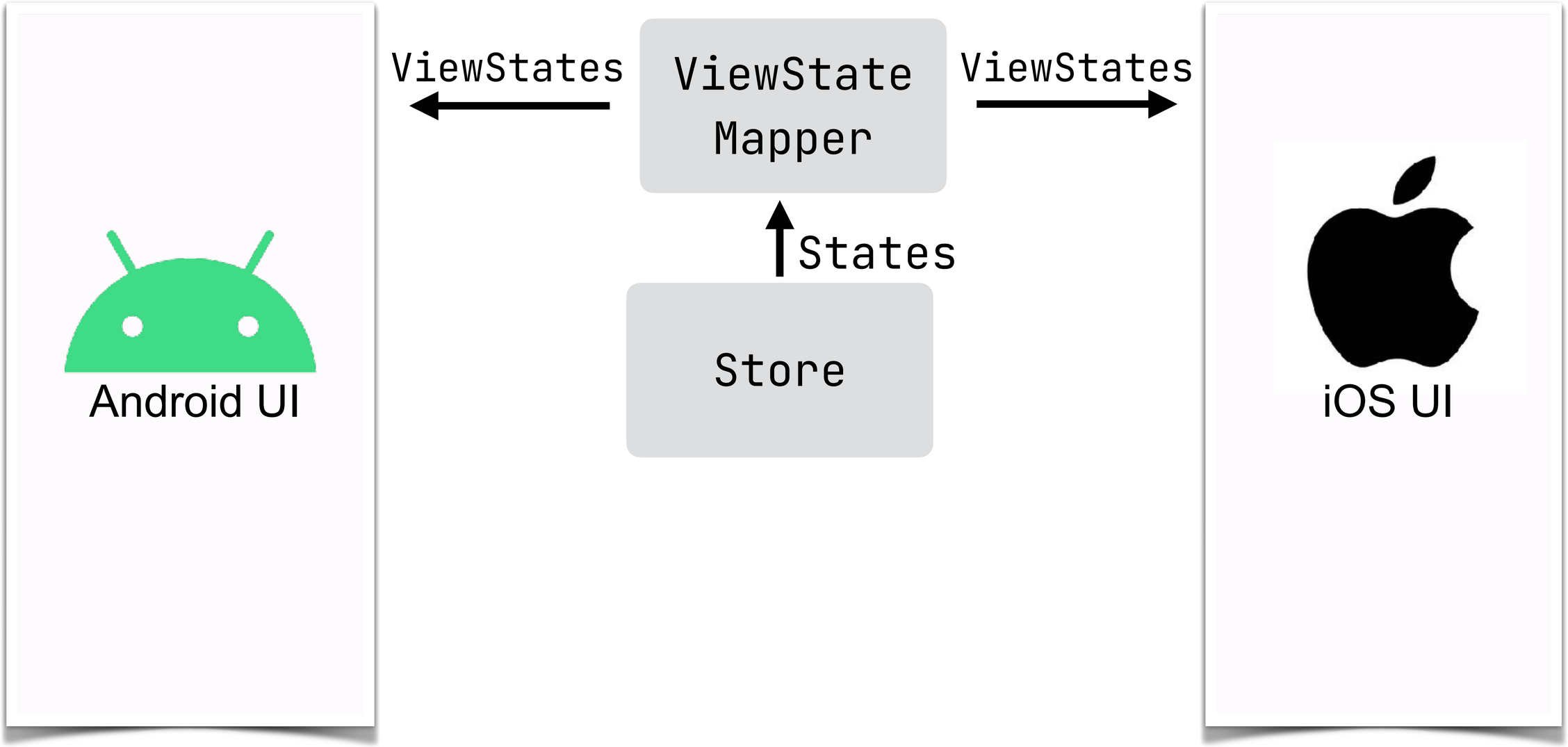
K/N → ObjC -> Swift interop problem

```
abstract class PLatformStream<T : Any>(private val publisher: Publisher<T>) {

    interface Publisher<T> {
        fun subscribe(subscriber: Subscriber<T>): Subscription
    }

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

    fun subscribe(subscriber: Subscriber<T>): Subscription {
        return publisher.subscribe(subscriber)
    }
}
```

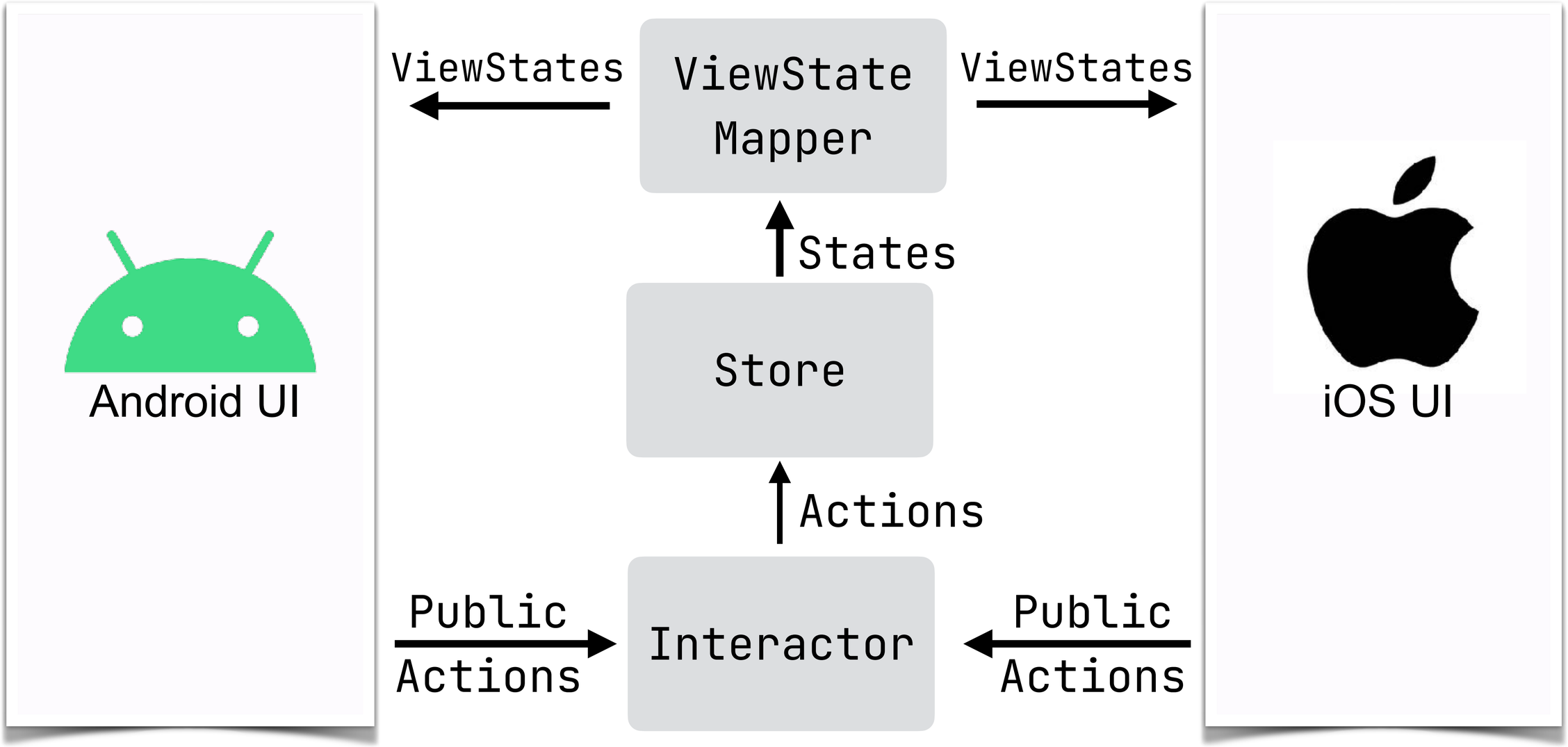
Abstract class interop OK

```
abstract class PlatformStream<T> → @interface PlatformStream<T>
```

Multiplatform ViewState + ViewStateMapping



Store encapsulated?



Naive feature module

```
— login-feature
    — androidMain
        — kotlin
            LoginFragment.kt
    — commonMain
        — kotlin
            LogingStore.kt
            LoginInteractor.kt
            LoginViewStateMapper.kt
            etc…
    — iosMain
        LoginViewController.swift
```

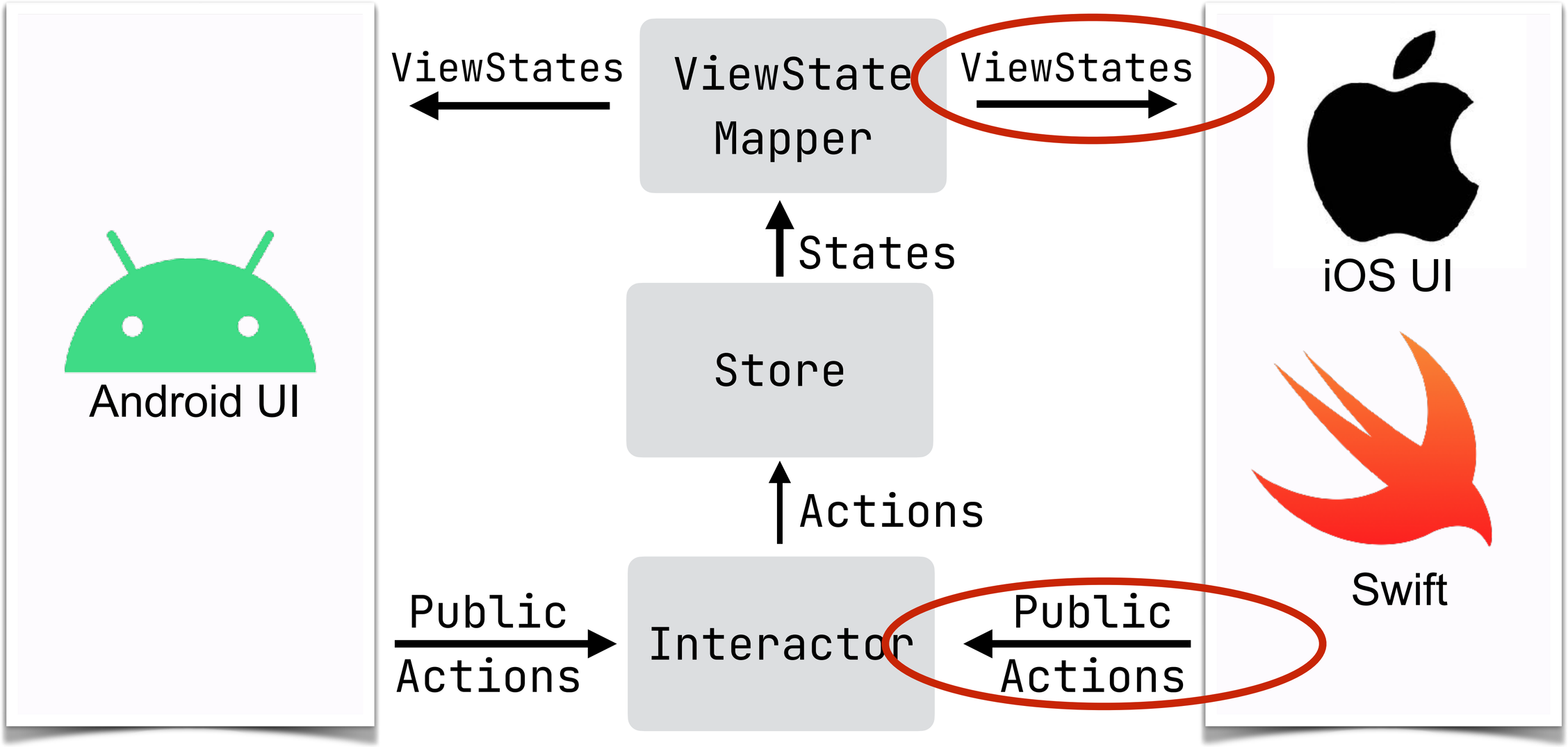
Store encapsulated?

```
class LoginFragment: Fragment() {
    @Inject
    lateinit var viewStateMapper: LoginViewStateMapper
    @Inject
    lateinit var interactor: LoginInteractor
    …

    override fun onViewCreated(…) {
        viewStateMapper.viewStates()
            .onEach { render(it) }
            .launchIn(scope)

        loginButton.onClick {
            interactor.dispatch(PerformLogin)
        }
    }
…
```

Store not encapsulated

```
class LoginFragment: Fragment() {
    @Inject
    lateinit var viewStateMapper: LoginViewStateMapper
    @Inject
    lateinit var interactor: LoginInteractor
    @Inject
    lateinit var store: Store<LoginState>
    …

    override fun onViewCreated(…) {
        viewStateMapper.viewStates()
            .onEach { render(it) }
            .launchIn(scope)

        loginButton.onClick {
            store.dispatch(PerformLogin)
        }
```

Implementation encapsulated?



Implementation encapsulated?

```
public class LoginViewController: UIViewController {

    override public func viewDidLoad() {
        super.viewDidLoad()
        setupViews()

        toRxSwift(viewStatesMapper.viewStates())
            .bind { [weak self] viewState in self?.render(viewState) }
            .disposed(by: lifetimeBag)

        loginButtonTaps
            .bind { [weak self] in
                self?.interactor.dispatch(action: PerformLogin())
            }.disposed(by: lifetimeBag)
    }

    private let interactor: LoginInteractor
    private let viewStateMapper: LoginViewStateMapper
    …
```

Implementation encapsulated?

```
public class LoginViewController: UIViewController {

    override public func viewDidLoad() {
        super.viewDidLoad()
        setupViews()

        toRxSwift(viewStatesMapper.viewStates())
            .bind { [weak self] viewState in self?.render(viewState) }
            .disposed(by: lifetimeBag)

        loginButtonTaps
            .bind { [weak self] in
                self?.interactor.dispatch(action: PerformLogin())
            }.disposed(by: lifetimeBag)
    }

    private let interactor: LoginInteractor
    private let viewStateMapper: LoginViewStateMapper
    …
```

Implemetation not encapsulated

```
internal class ViewState(…)
internal interface LoginViewStateMapper{…}
internal interface LoginInteractor{…}
```

Everything not encapsulated

Not needed by UI but visible to it

```
internal class LoginState(…)
internal typealias LoginStore = Store<LoginState>
internal class SomeEpic(…)
Database
Network
…
```

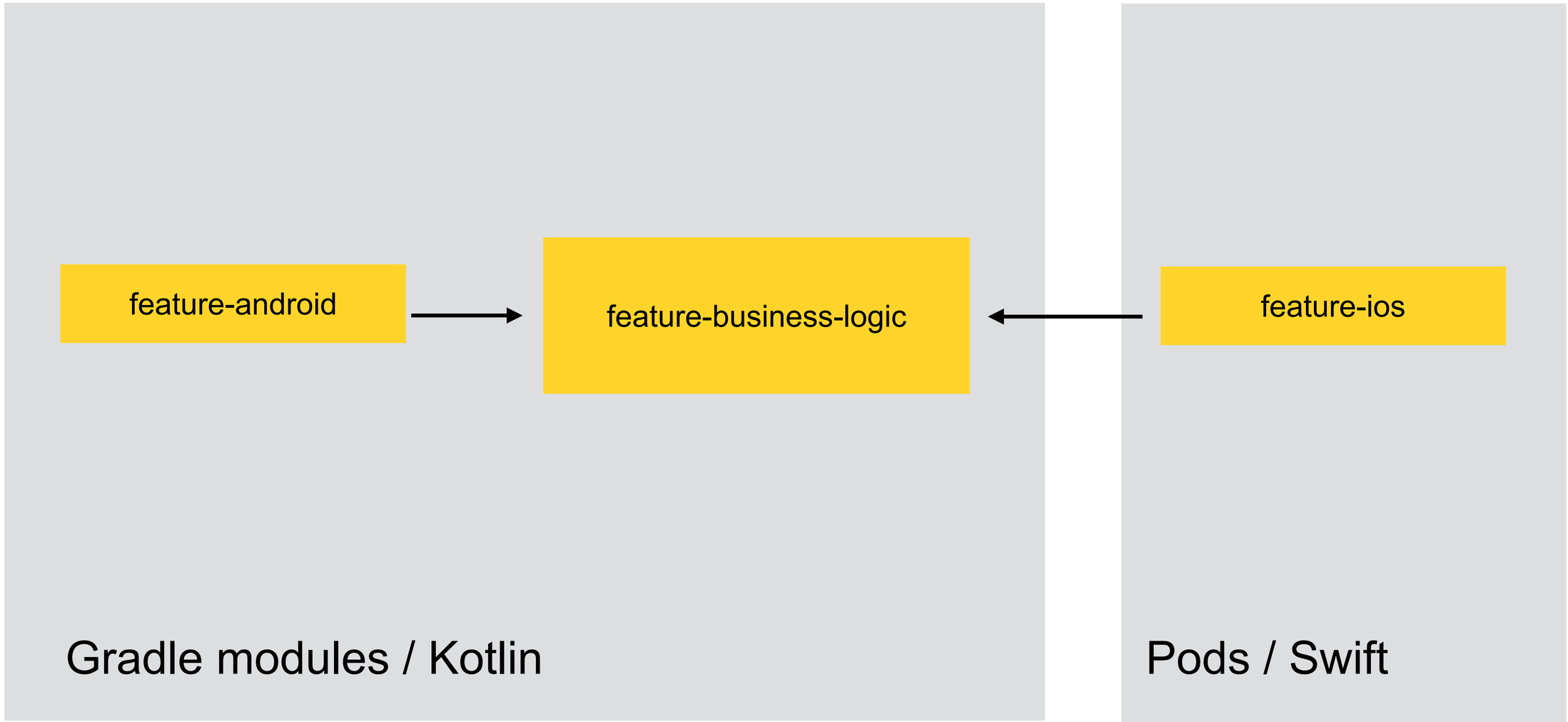
Visible in client

```
~~internal~~ class ViewState(…)
~~internal~~ interface LoginViewStateMapper{…}
~~internal~~ interface LoginInteractor{…}
```

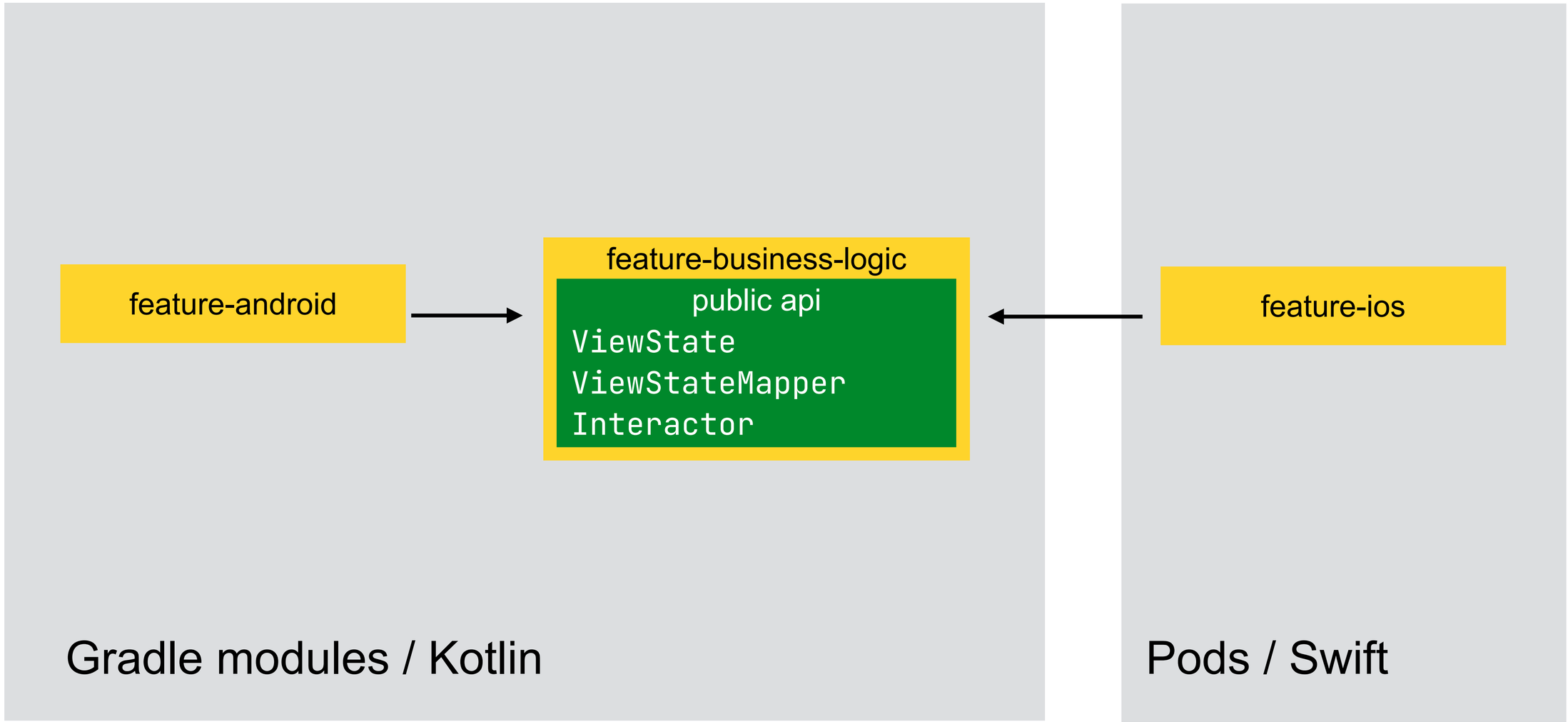
UI must not see logic details

```
class LoginFragment: Fragment() {
    @Inject
    lateinit var viewStateMapper: LoginViewStateMapper
    @Inject
    lateinit var interactor: LoginInteractor
    @Inject
    lateinit var store: Store<LoginState>
    …

    override fun onViewCreated(…) {
        viewStateMapper.viewStates()
            .onEach { render(it) }
            .launchIn(scope)

        loginButton.onClick {
            store.dispatch(PerformLogin)
        }
```

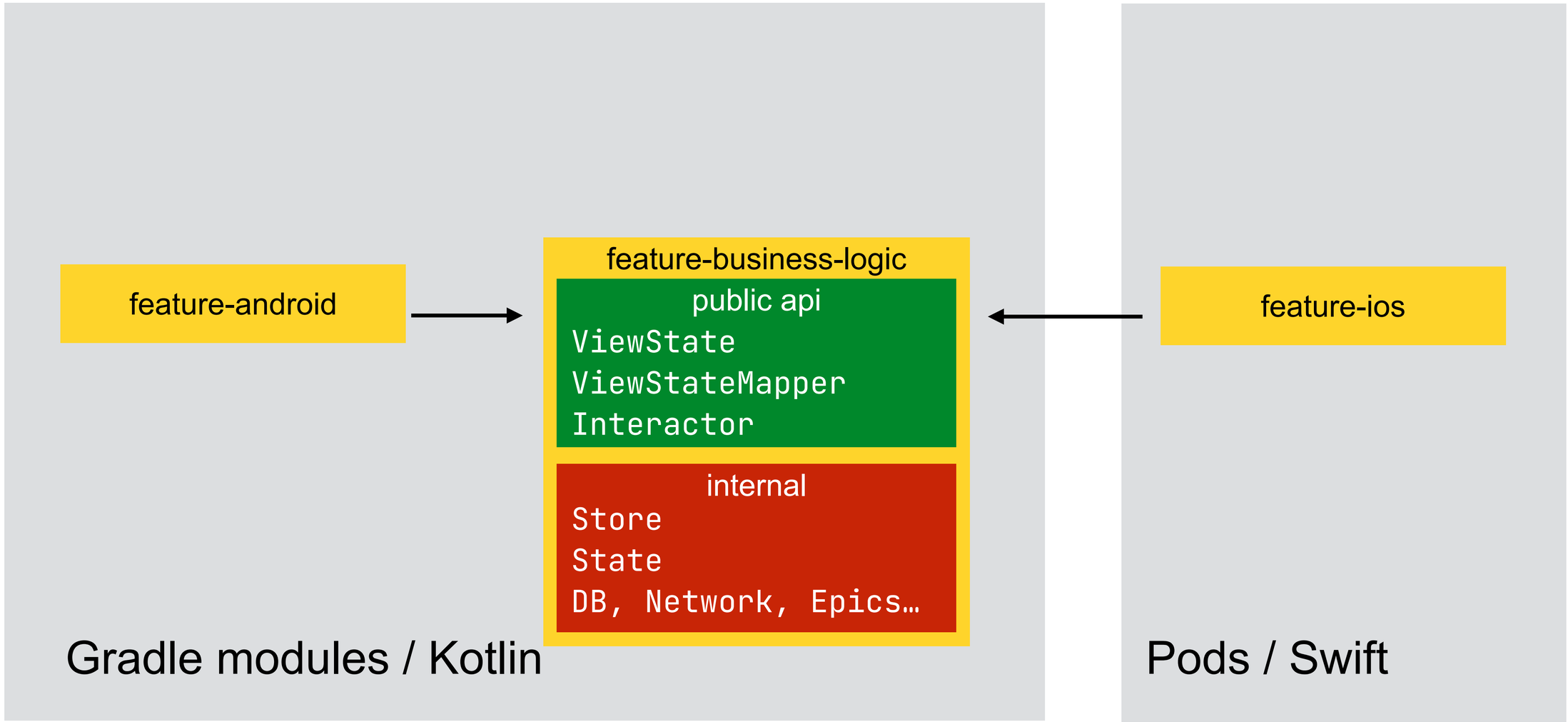
Split logic and UI



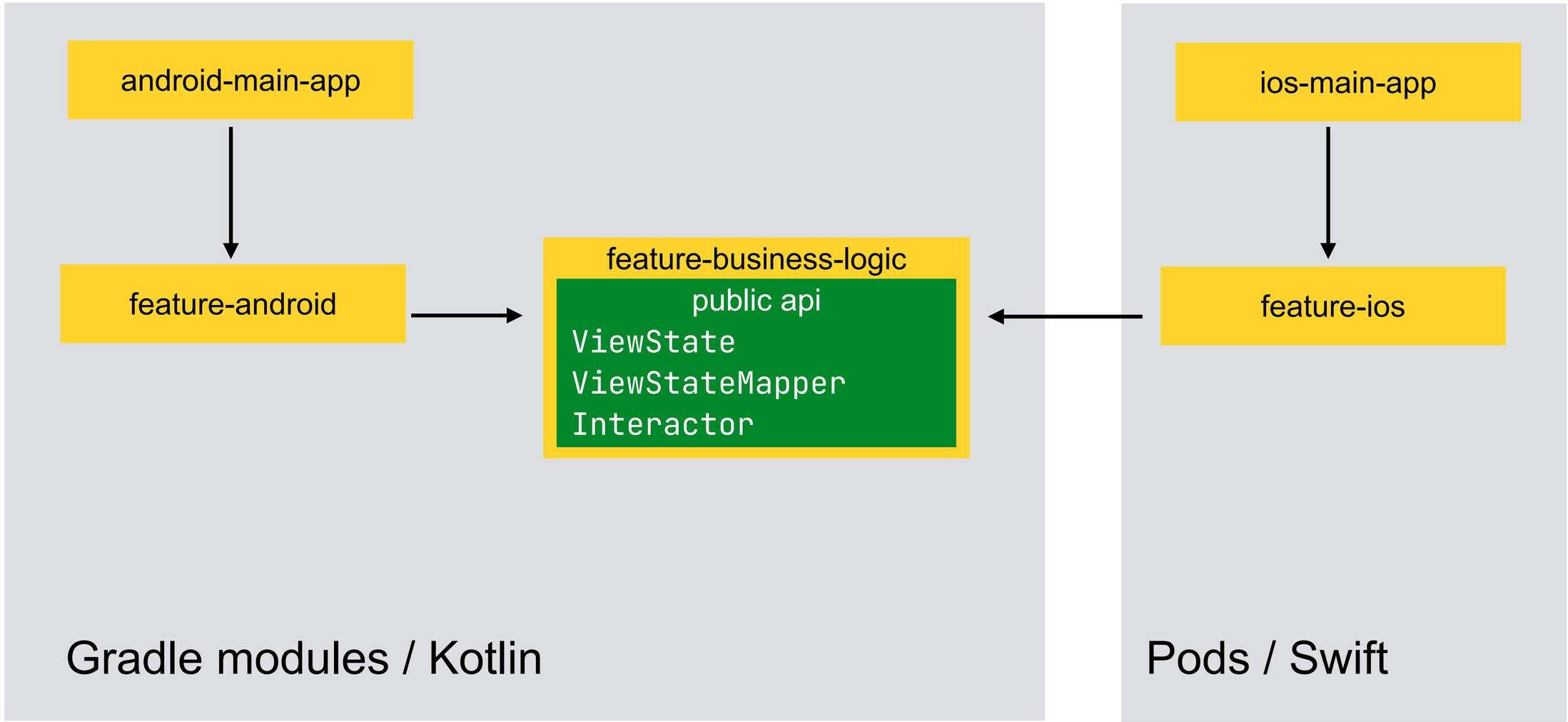
Split logic and UI



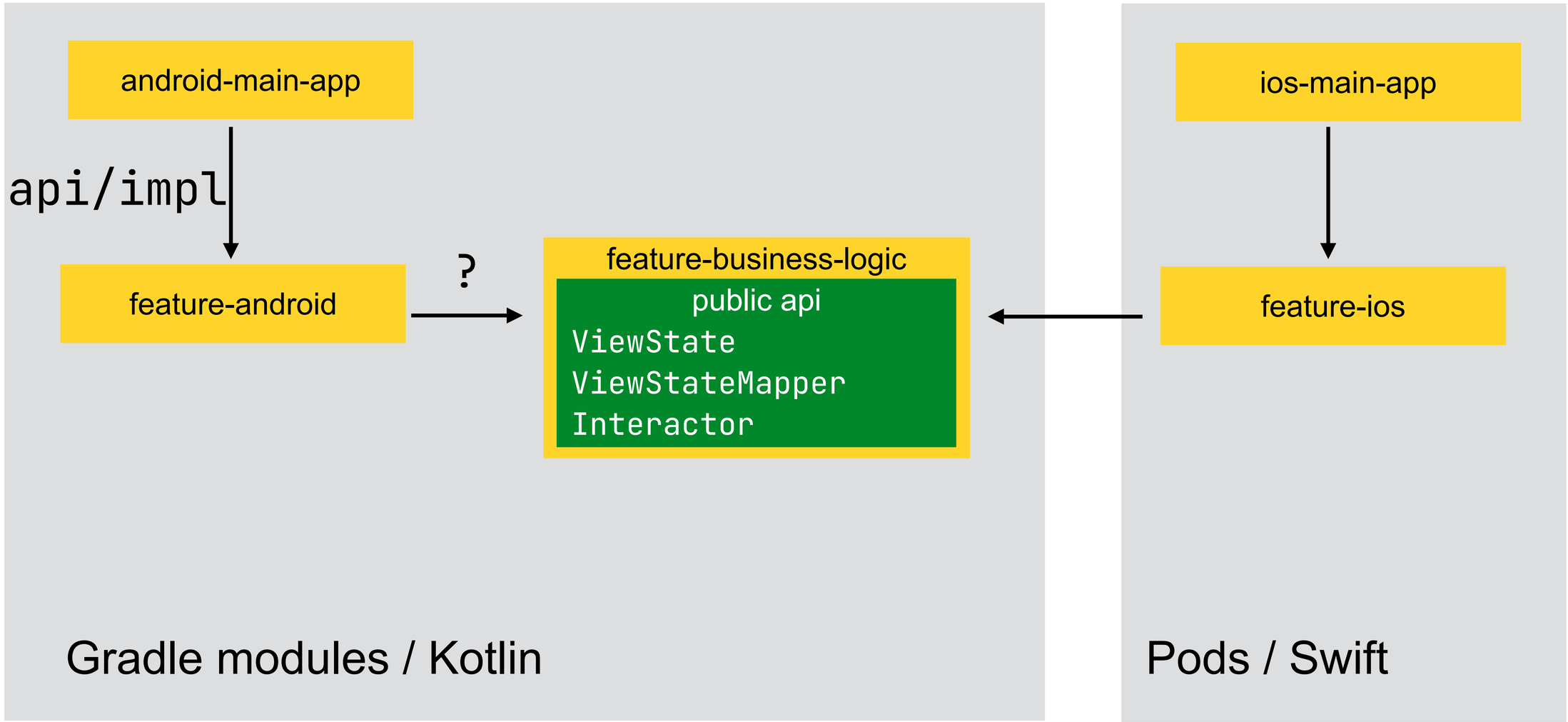
Logic details encapsulated now!



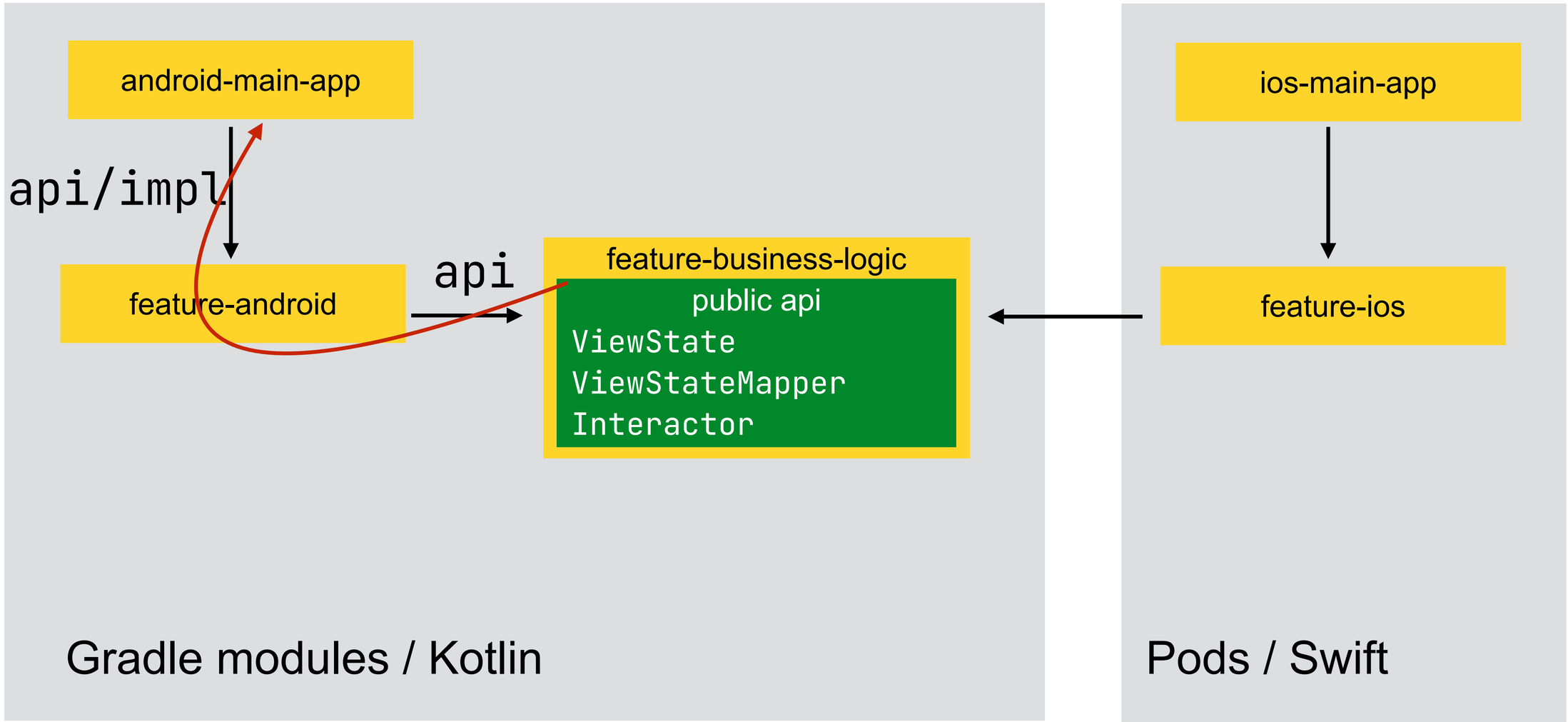
What about client?



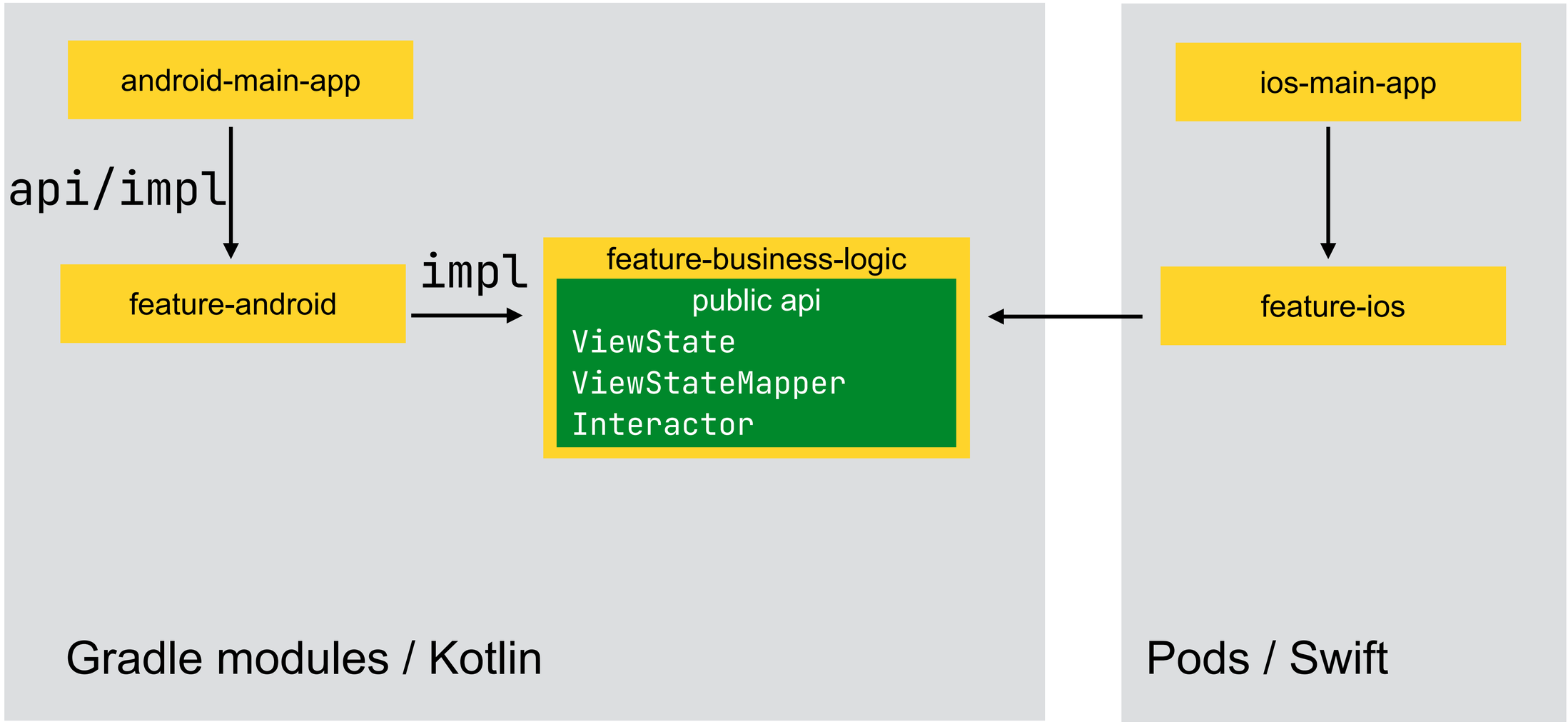
api or implementation?



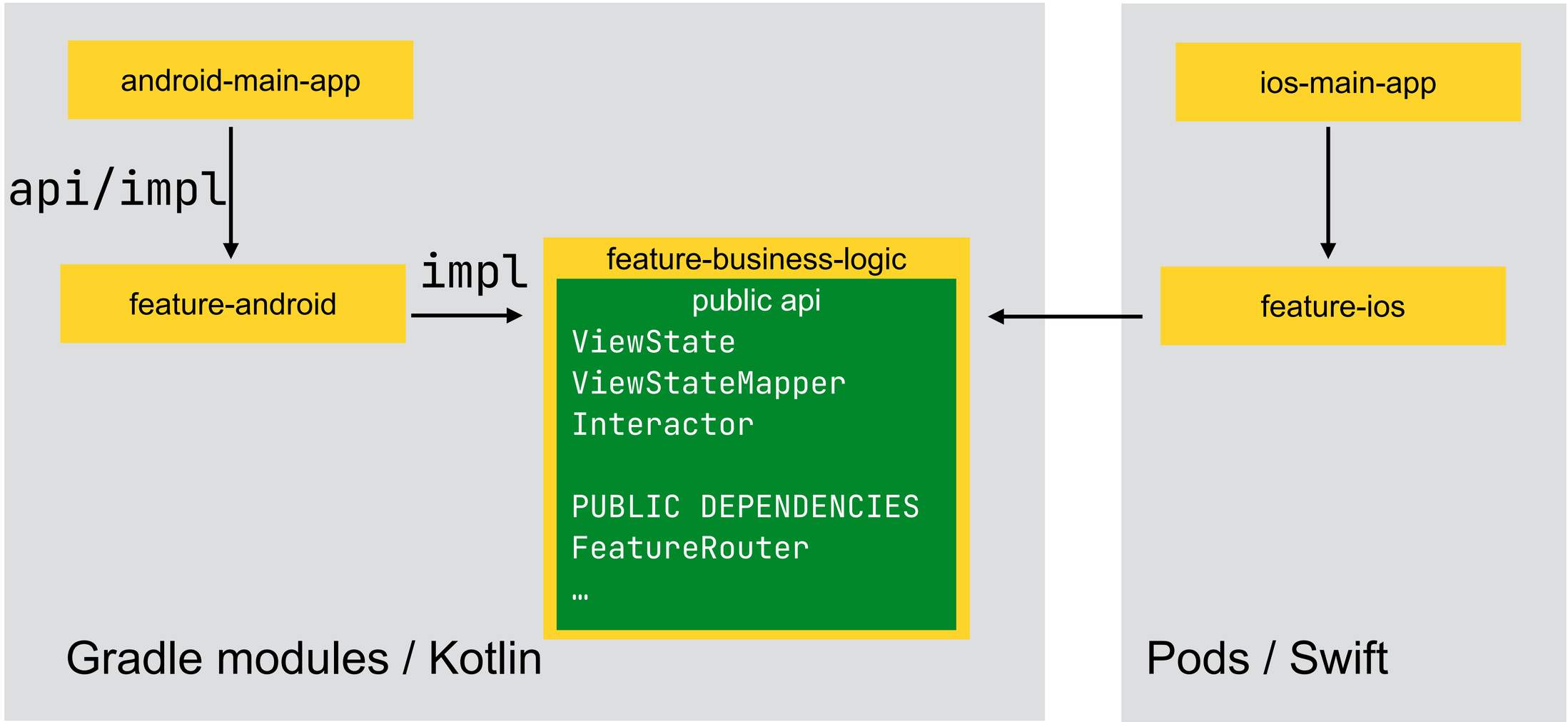
Not api!


android-main-app
api/impl
feature-android
api
feature-business-logic
public api
ViewState
ViewStateMapper
Interactor
ios-main-app
feature-ios
Gradle modules / Kotlin
Pods / Swift

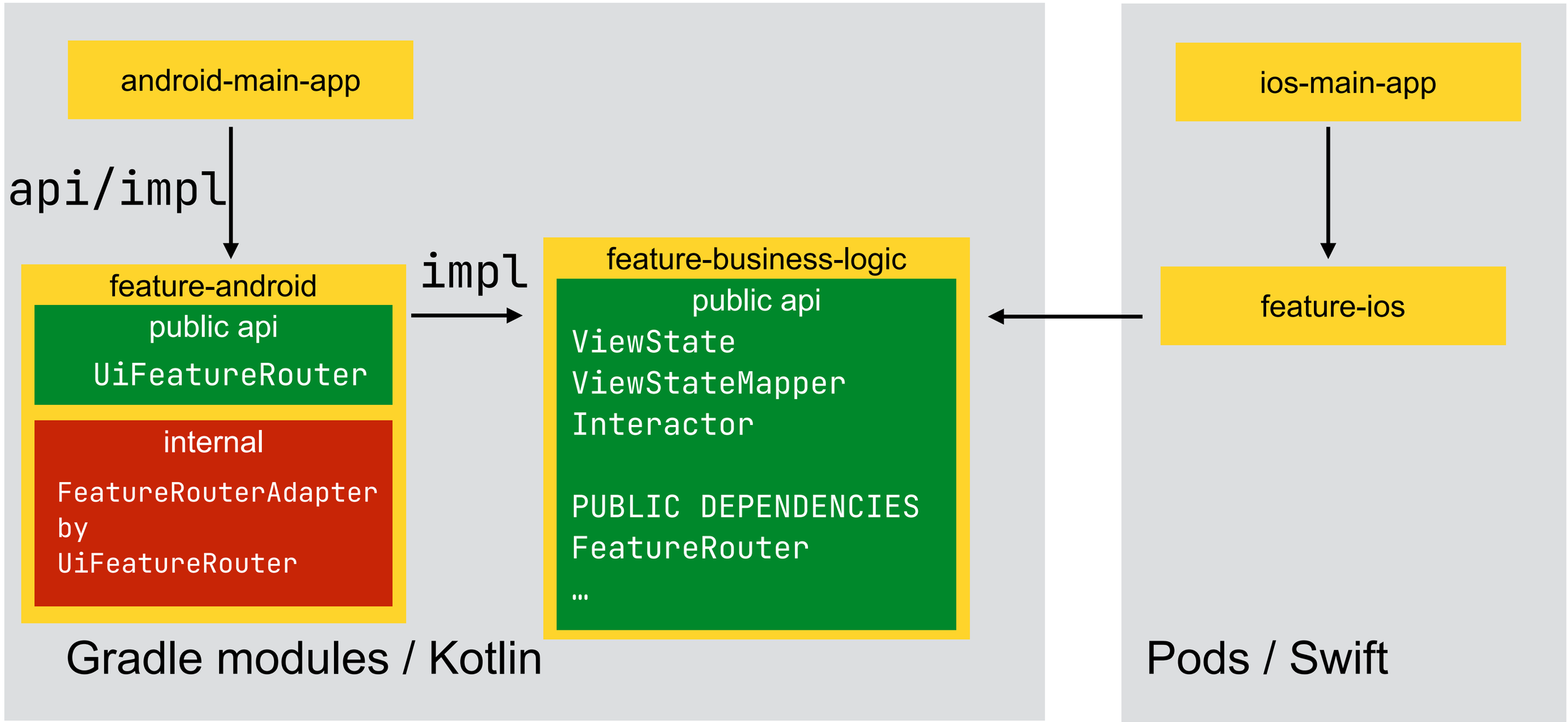
implementation?



Public dependencies



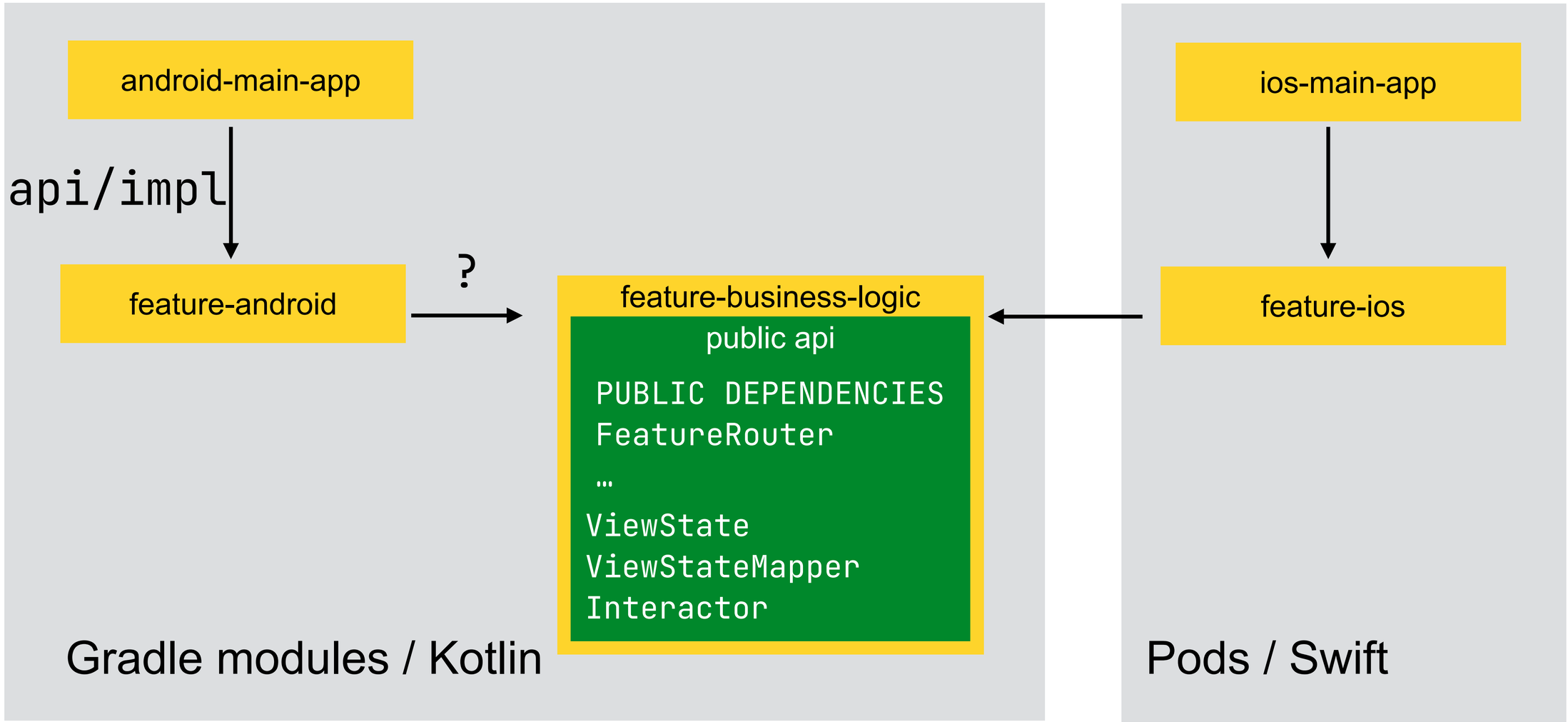
Boilerplate and api duplication



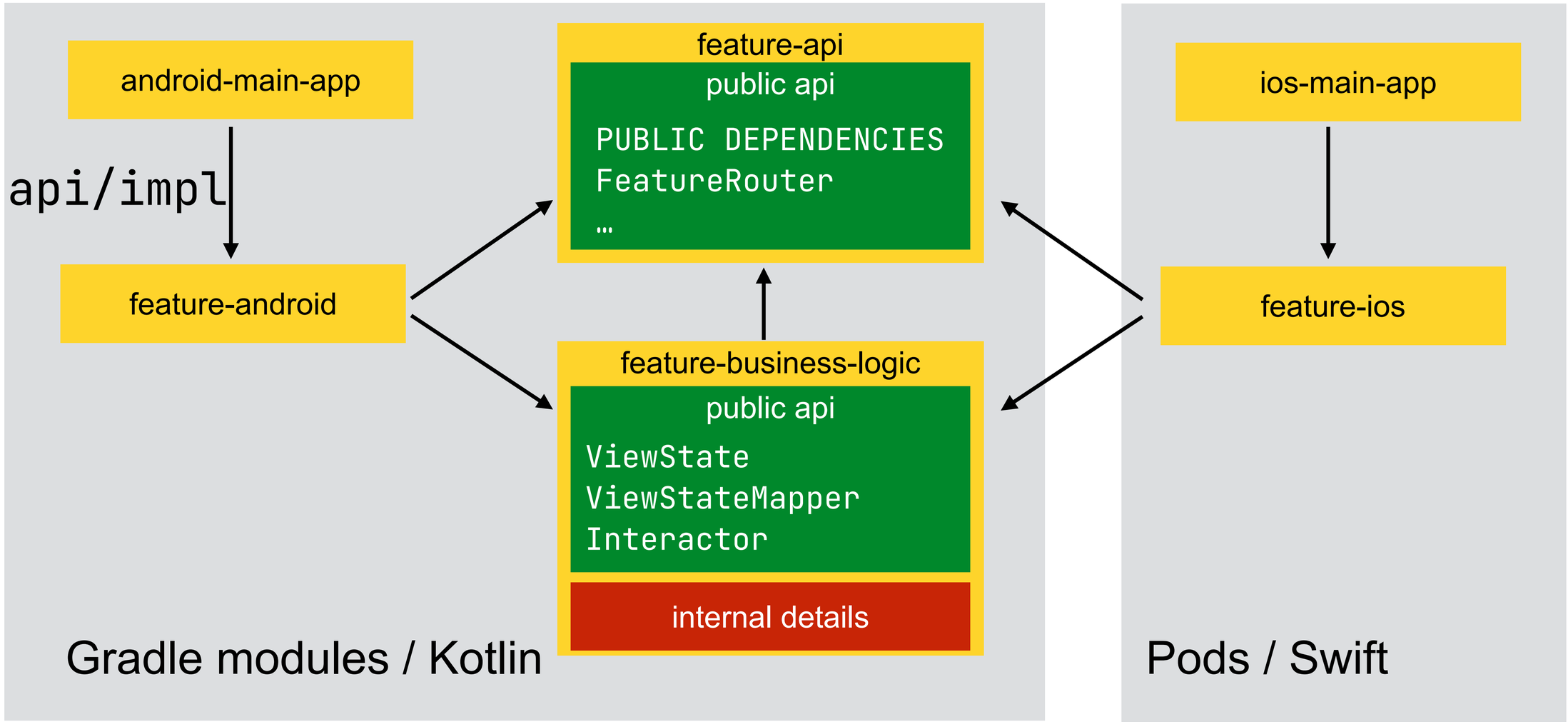
Boilerplate and api duplication

```
//:feature-business-logic
interface FeatureRouter {
    fun openSomething(someArg: String)
    fun openSomethingElse(other: String)
}

//:feature-android
interface UiFeatureRouter {
    fun openSomething(someArg: String)
    fun openSomethingElse(other: String)
}

internal class FeatureRouterAdapter(
    private val router: UiFeatureRouter,
): FeatureRouter {
    override fun openSomething(someArg: String) {
        router.openSomething(someArg = someArg)
    }
   override fun openSomethingElse(other: String) {
        router.openSomethingElse(other = other)
    }
}
```

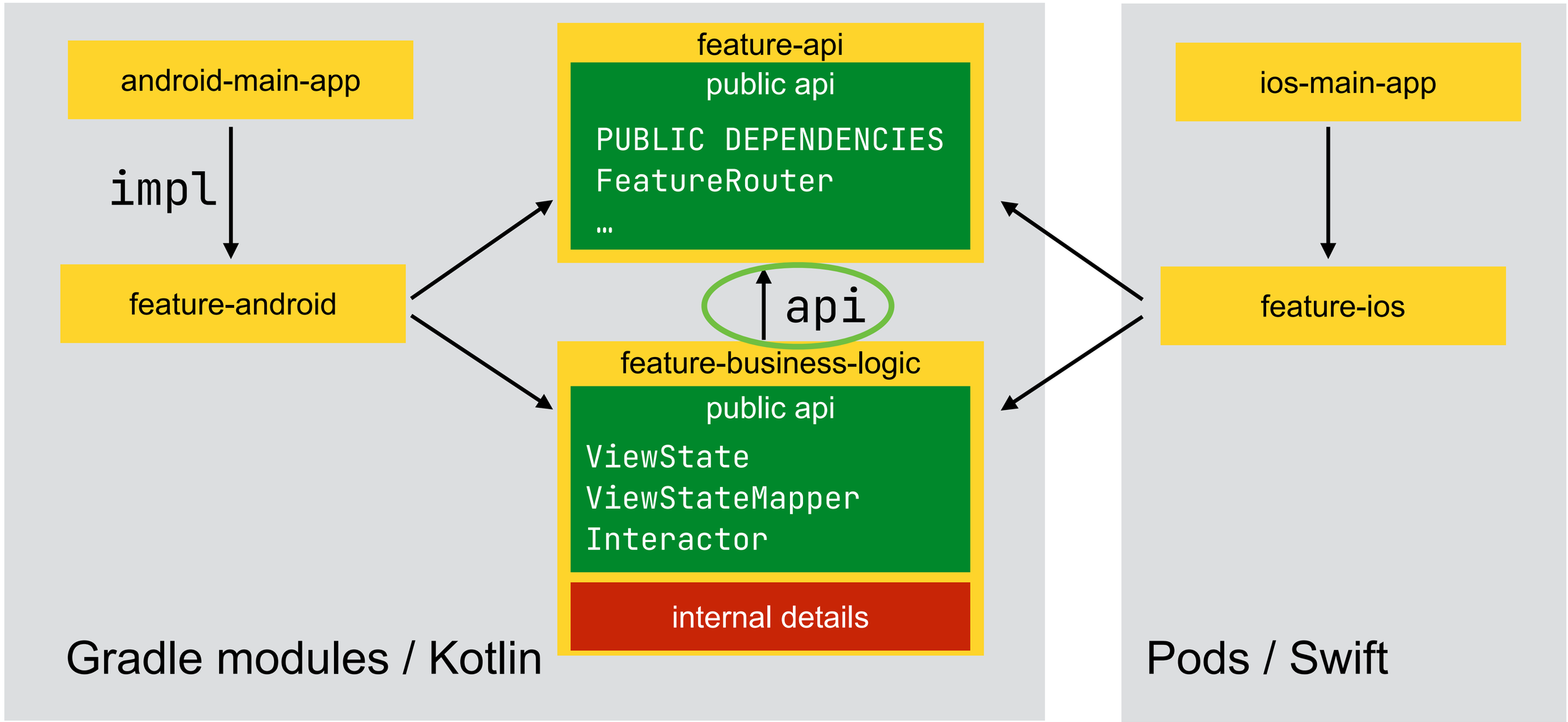
How to depend?



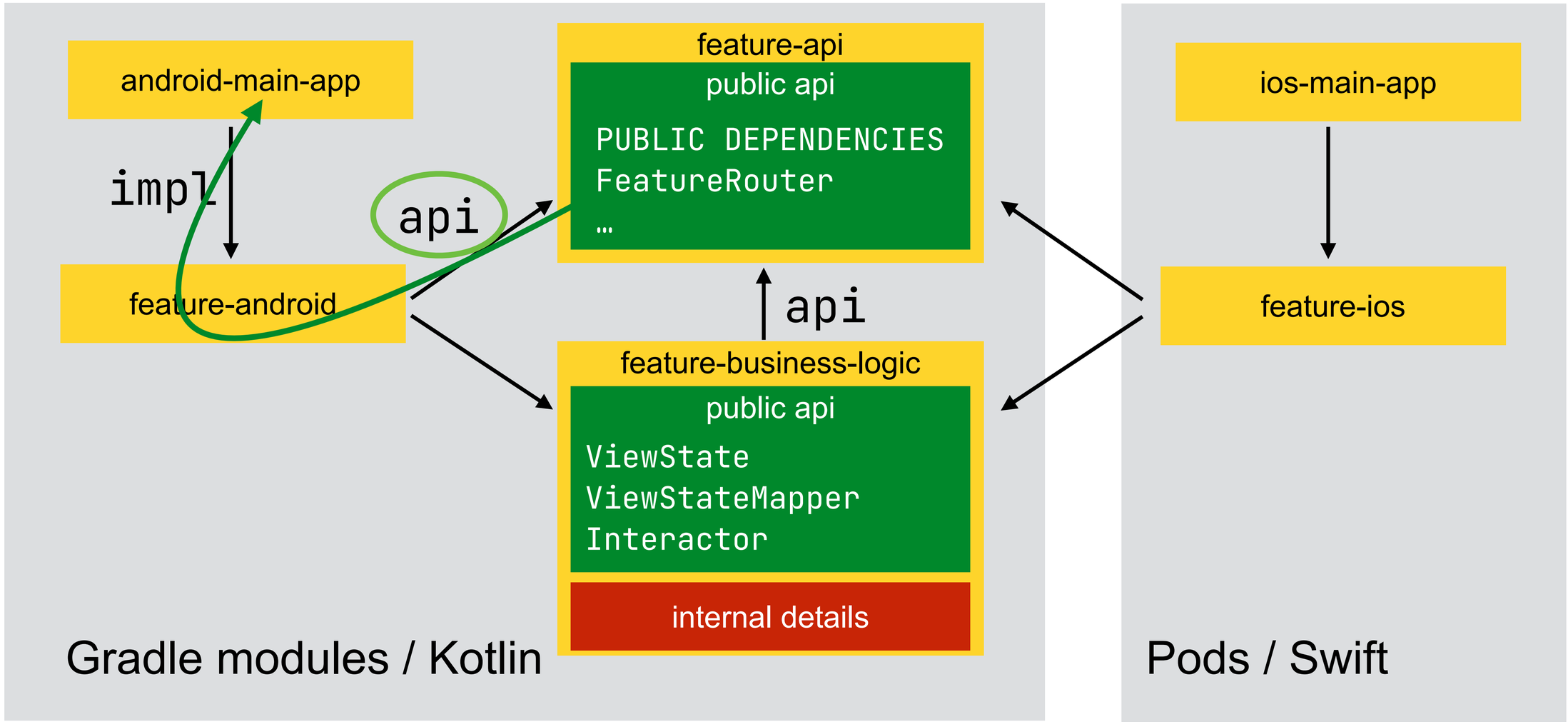
Split to api and implementation



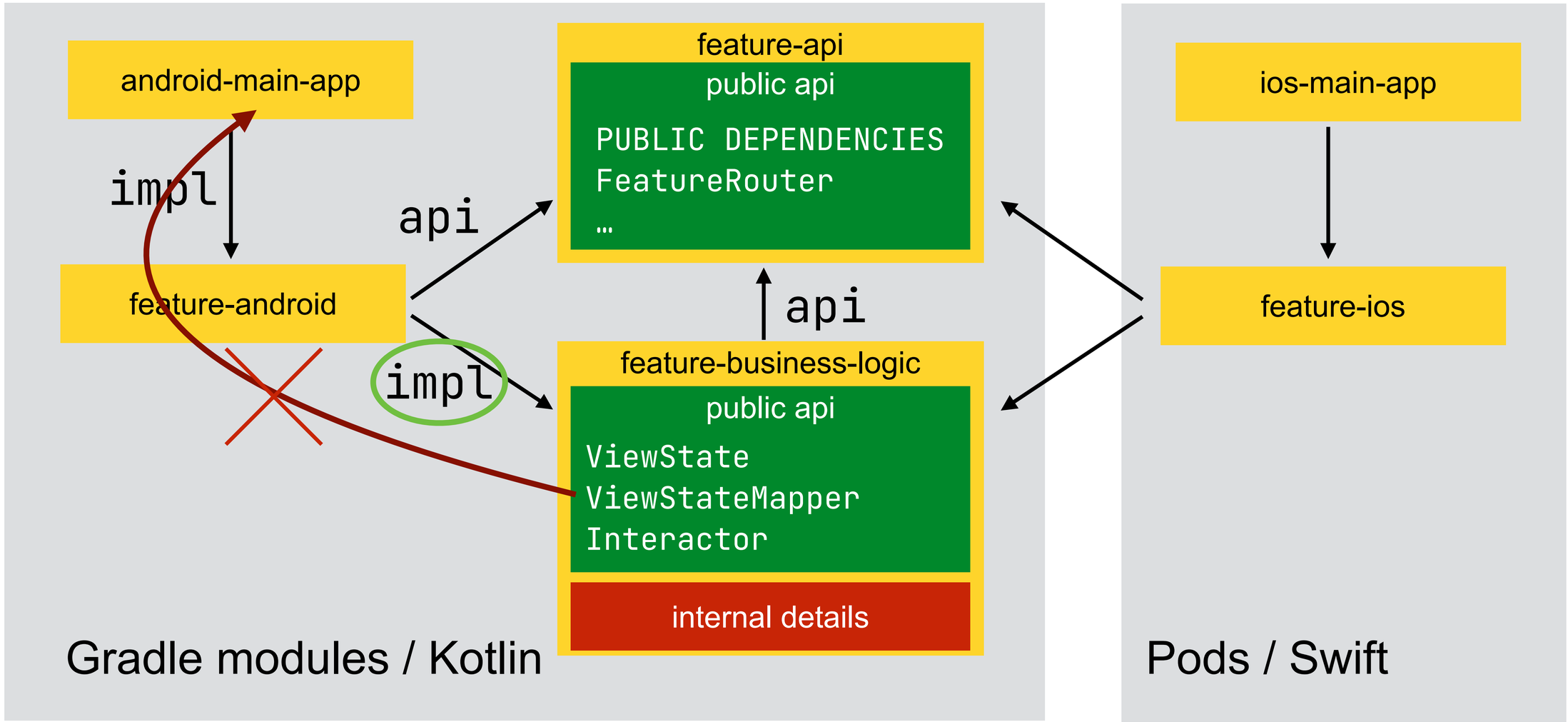
Logic depends on api



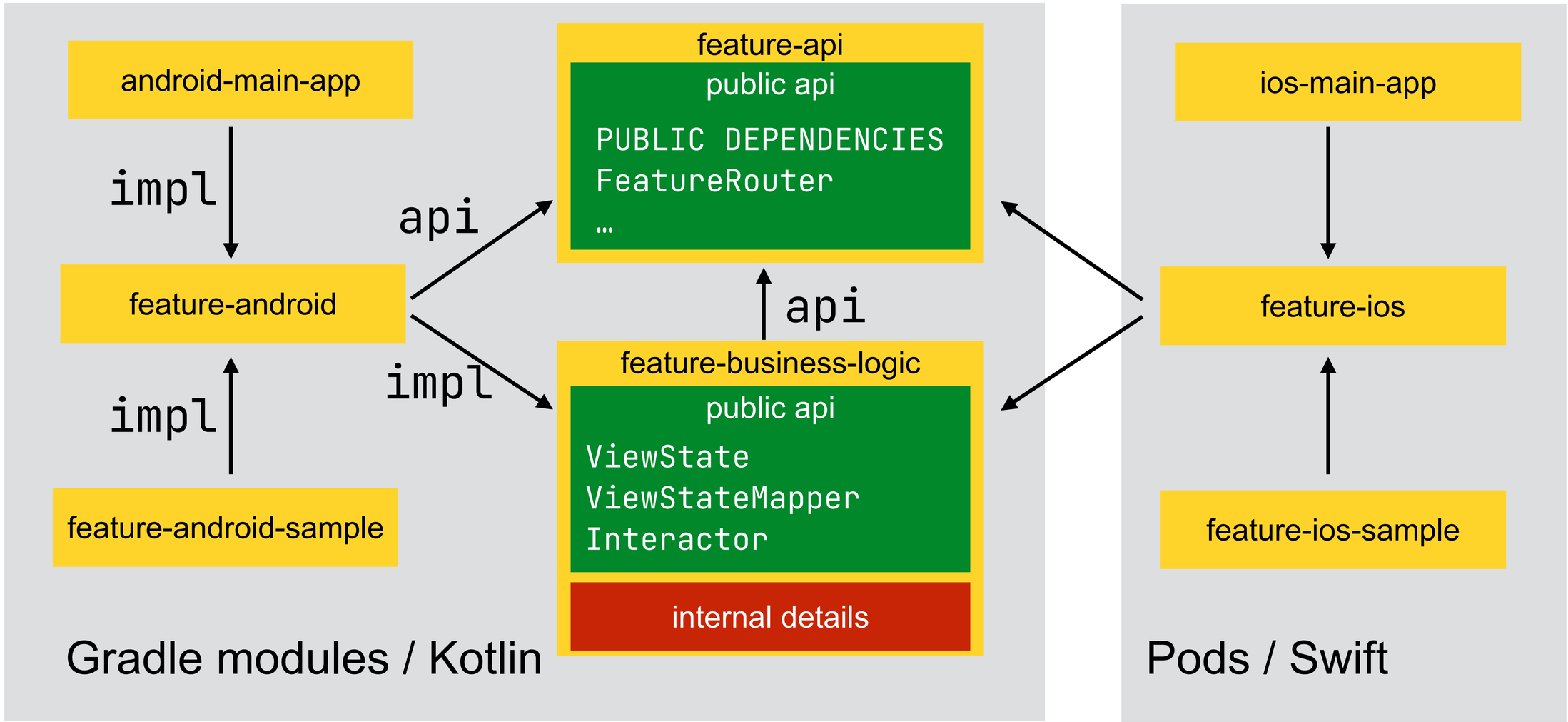
Api visible to client



‘Internal’ api not visible to client



Feature modules structure



Feature module dependencies definition

```
interface FeatureDependencies {
    val someService: SomeService
    val externalRouter: ExternalRouter
}

interface SomeService {
    fun doSomething()
}

interface ExternalRouter {
    fun openWebView(url: String)
}
```

Glue it together: Kinzhal

- DI library
- Compile-time
- Multiplatform(KSP)
- Looks like Dagger

https://github.com/daugeldauge/kinzhal

Looks like Dagger

```
@Scope
internal annotation class FeatureScope

@FeatureScope
@Component(
    modules = [
        MyModule::class
    ],
    dependencies = [
        FeatureDependencies::class,
    ],
)
internal interface FeatureComponent {
    fun interactor(): FeatureInteractor
    fun viewStatesMapper(): FeatureViewStatesMapper
}
```

https://github.com/daugeldauge/kinzhal

Redux tips

- **Look at State**
- Reduce field by field
- No harmful Epics

Redux tips

- Look at State
- **Reduce field by field**
- No harmful Epics

How to reduce properly?

```
data class CheckoutState(
    val cost: Float,
    val city: String? = null,
    val street: String? = null,
    val postalCode: String? = null,
    val phone: String? = null,
)
```

How to reduce properly?

```
data class CheckoutState(
    val cost: Float,
    val city: String? = null,
    val street: String? = null,
    val postalCode: String? = null,
    val phone: String? = null,
)

class ChangeCity(val city: String?) : Action
class ChangePhone(val phone: String?) : Action
class ChangeStreet(val street: String?) : Action
class ChangePostalCode(…): Action
```

Reduce action by action?

```
fun reduce(state: CheckoutState, action: Action): CheckoutState {
    return when (action) {
        is ChangeCity → state.copy(city = action.city)
        is ChangePhone → state.copy(phone = action.phone)
        is ChangeStreet → state.copy(street = action.street)
        is ChangePostalCode → state.copy(postalCode = action.code)
        else → state
    }
}
```

State refactoring

```
data class CheckoutState(
    val cost: Float,
    val address: Address = Address(),
    val phone: String? = null,
)

data class Address(
    val city: String? = null,
    val street: String? = null,
    val postalCode: String? = null,
)
```

Reduce action by action — wrong!

```
fun reduce(state: CheckoutState, action: Action): CheckoutState {
    return when (action) {
        is ChangeCity → state.copy(
            address = state.address.copy(
                city = action.city,
            )
        )
        …
        is ChangePhone → state.copy(phone = action.phone)

        else → state
    }
}
```

Reduce field by field

```
fun reduce(state: CheckoutState, action: Action): CheckoutState {
    return state.copy(
        city = reduceCity(state.city, action),
        street = reduceStreet(state.street, action),
        postalCode = reducePostalCode(…),
        phone = reducePhone(…),
    )
}

private fun reducePhone(phone: String?, action: Action) = when (action) {
    is ChangePhone → action.phone
    else → phone
}
private fun reduceStreet(street: String?, action: Action) = when (action)…
private fun reduceCity(…) = …
private fun reducePostalCode(…) = …
```

Reduce refactored state

```
fun reduce(state: CheckoutState, action: Action): CheckoutState {
    return state.copy(
        address = reduceAddress(state.address, action),
        phone = reducePhone(state.phone, action),
    )
}

private fun reduceAddress(address: Address, action: Action): Address {
    return when (action) {
        is ChangeCity → address.copy(city = action.city)
        is ChangeStreet → …
        …
        else → phone
    }
}
```

One more time: field by field!

```
fun reduce(state: State, action: Action): State {
    return state.copy(
        field1 = reduceField1(state.field1, action),
        field2 = reduceField2(state.field2, action),
        …,
        fieldM = reduceFieldM(state.fieldM, action),
    )
}
```

```
fun reduce(state: State, action: Action): State {
    return when (action) {
        is Action1 → state.copy(field1 = action.data)
        is Action2 → state.copy(…)
        …
        is ActionN → state.copy(…)
        else → state
    }
}
```

Redux tips

- Look at State
- Reduce field by field
- **No harmful Epics**

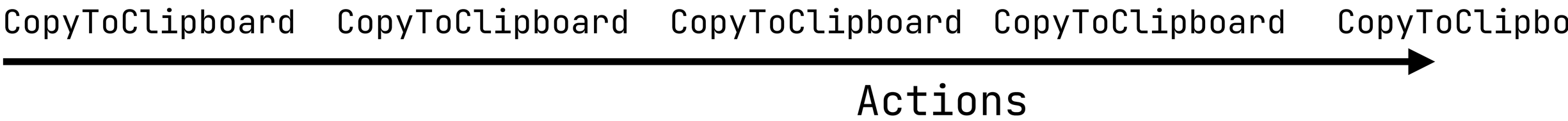
Epic must consume actions!

Epic must consume actions!

Epic must consume actions!

Epic must consume actions!

```
class HarmfulClipboardEpic @Inject constructor(
    val clipboard: Clipboard,
): Epic() {
    override fun act(actions: Flow<Action>): Flow<Action> {
        return actions
            .ofType<CopyToClipboard>()
            .onEach { clipboard.copy(text = it.text) }   }
}
```



Epic must consume actions!

```
class ClipboardEpic @Inject constructor(
    val clipboard: Clipboard,
): Epic() {
    override fun act(actions: Flow<Action>): Flow<Action> {
        return actions
            .ofType<CopyToClipboard>()
            .onEach { clipboard.copy(text = it.text) }   }
            .skipAll()
}
```

Epic must consume actions!

```
class ClipboardEpic @Inject constructor(
    val clipboard: Clipboard,
): Epic() {
    override fun act(actions: Flow<Action>): Flow<Action> {
        return actions
            .ofType<CopyToClipboard>()
            .onEach { clipboard.copy(text = it.text) }   }
            .map { NotifyAction("Copied to clipboard") }
}
```

?