SERGEY SHULGA

MANAGING STATE IN REACTIVE APPLICATIONS

STATE

MVVM

FRP

PROBLEMS

- A big team without an established pattern of how to use it

PROBLEMS

- A big team without an established pattern of how to use it
- different understanding of RP

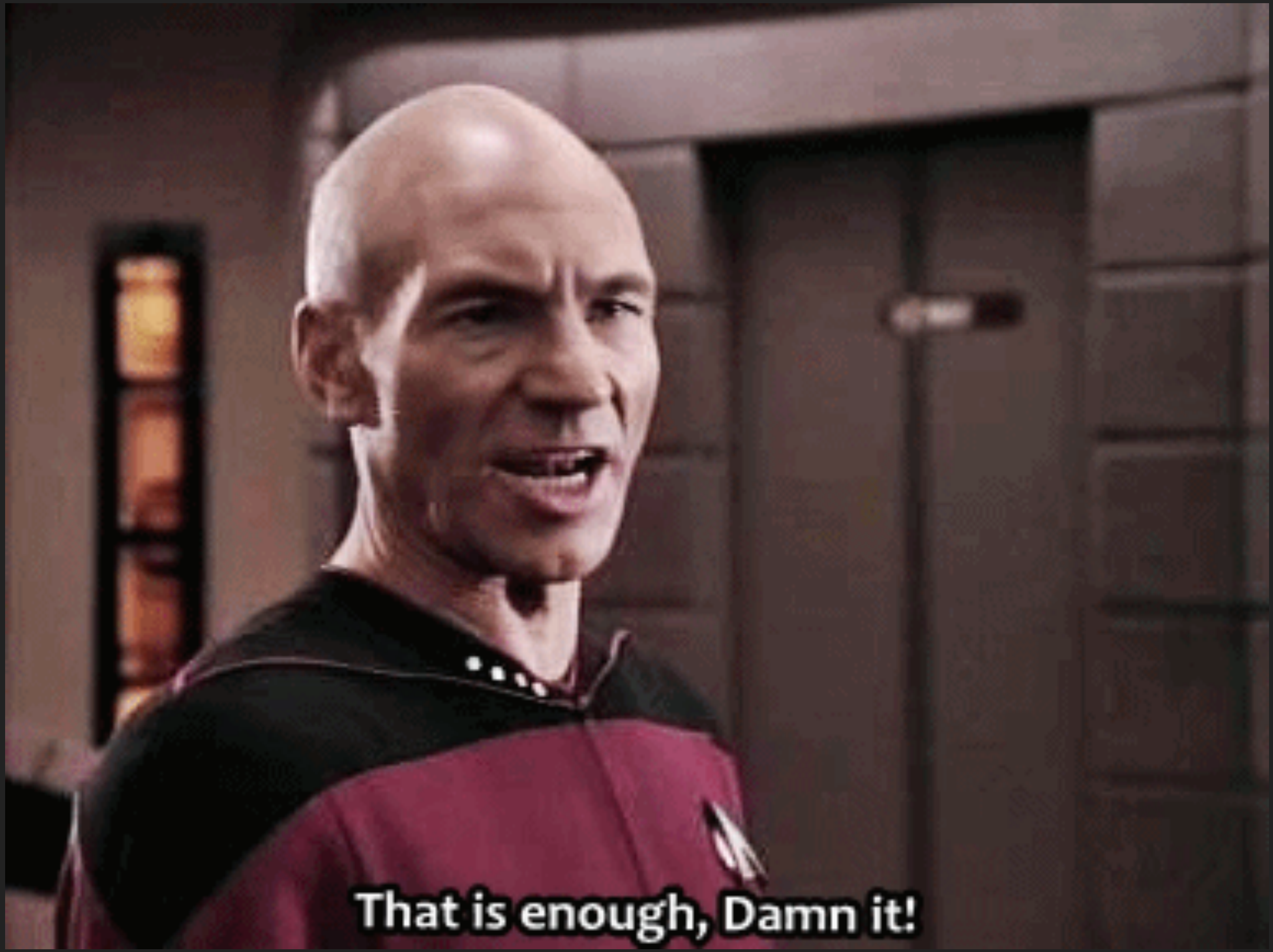
PROBLEMS

- A big team without an established pattern of how to use it
- different understanding of RP
- different perspectives of how these principles should be applied

PROBLEMS

- A big team without an established pattern of how to use it
- different understanding of RP
- different perspectives of how these principles should be applied
- different level of knowledge/experience

ENOUGH



DIFFERENT STYLES

DIFFERENT STYLES

- Kickstarter style. Input/Output protocols

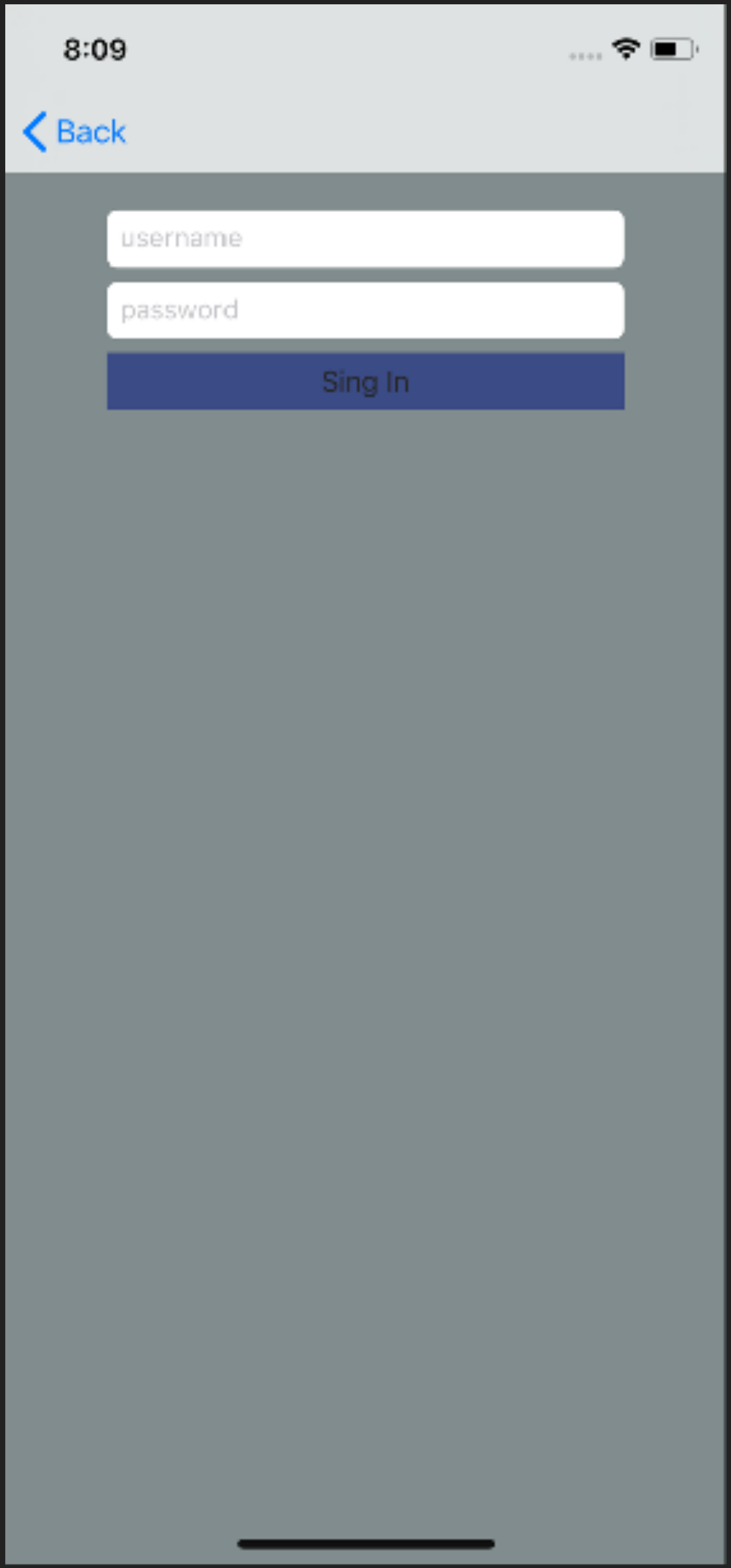
DIFFERENT STYLES

- Kickstarter style. Input/Output protocols
- (Input/Output) are observables

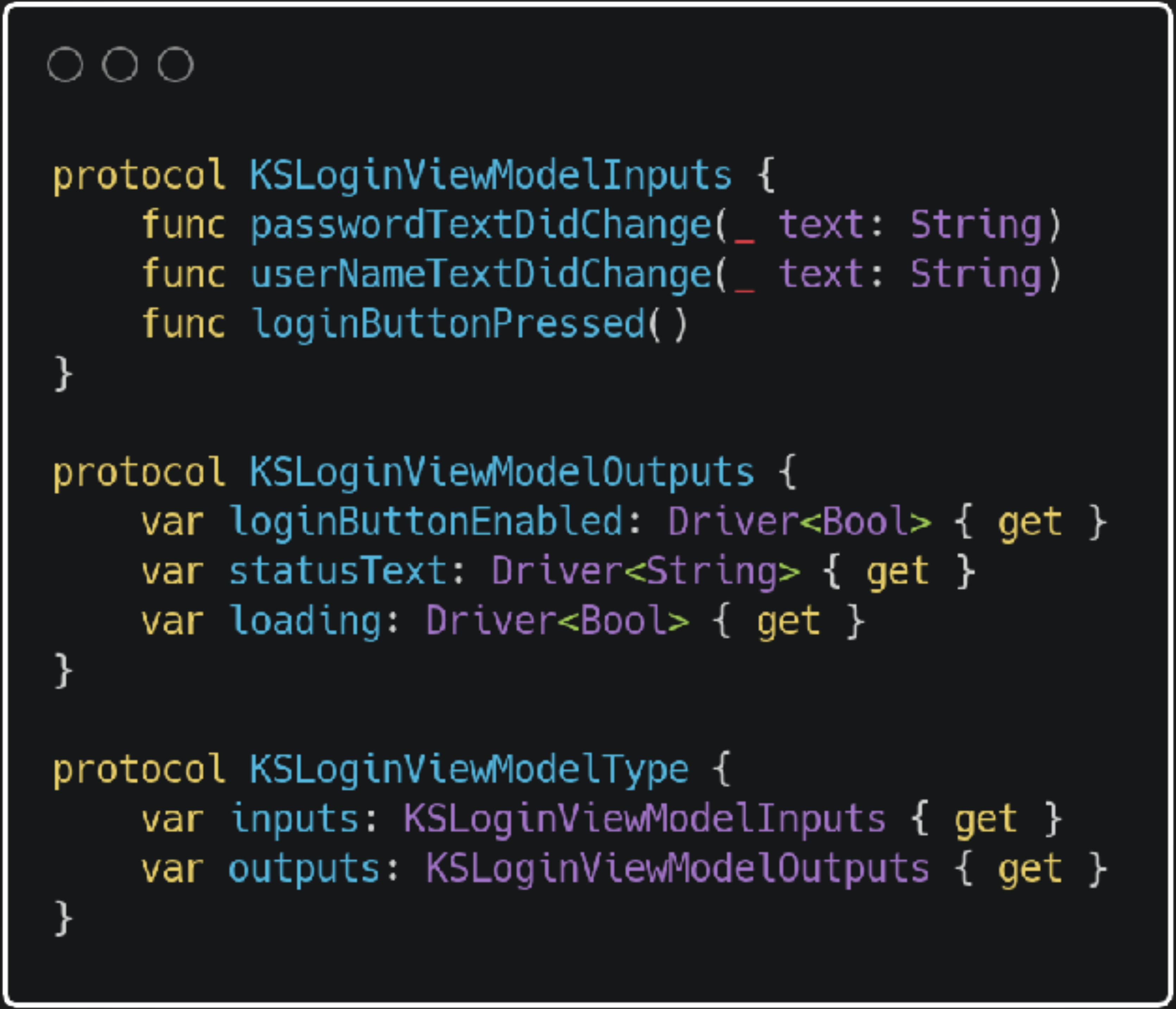
DIFFERENT STYLES

- Kickstarter style. Input/Output protocols
- (Input/Output) are observables
- Using Variables/Subjects/Relays

EXAMPLE

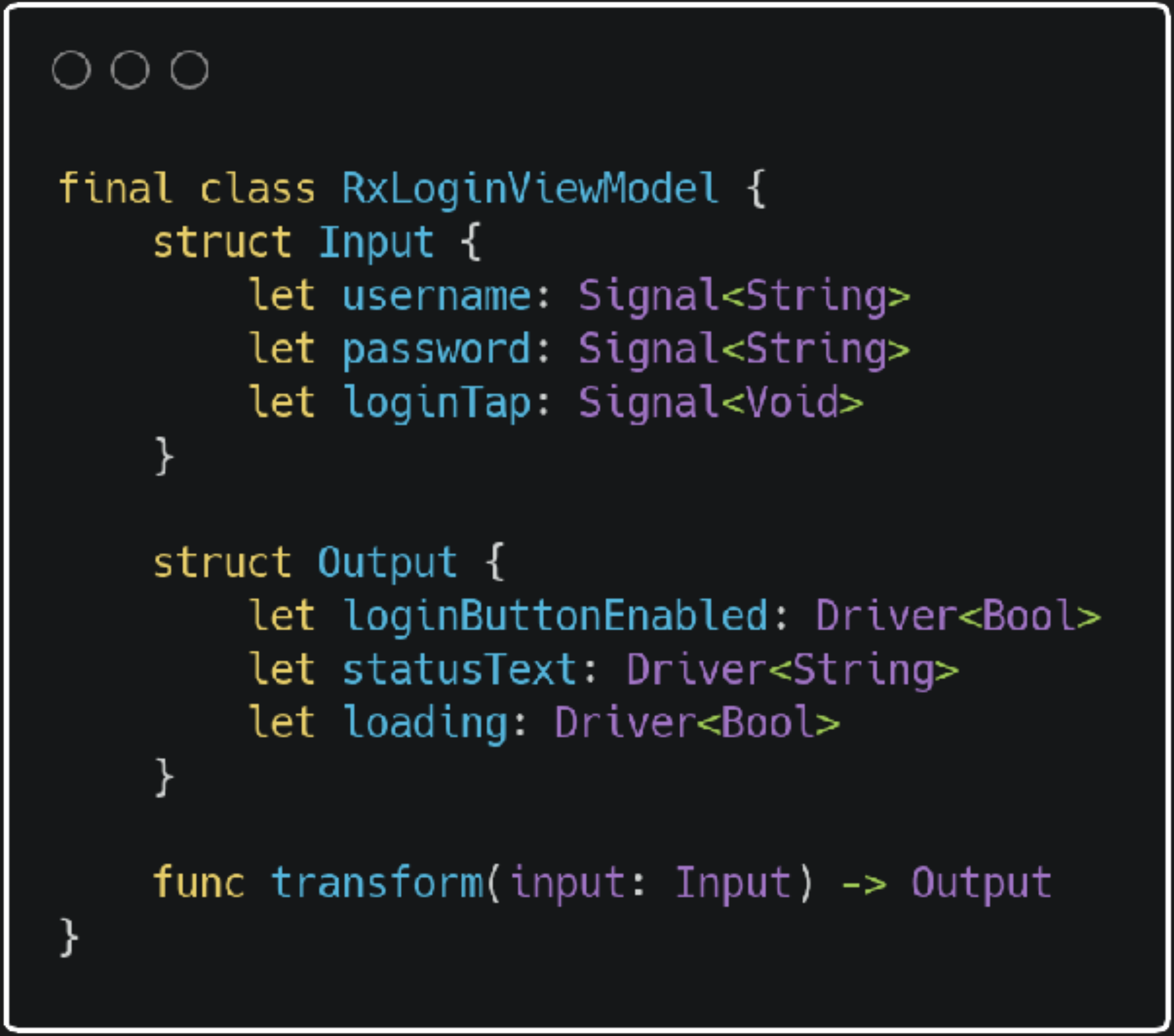


KICKSTARTER STYLE



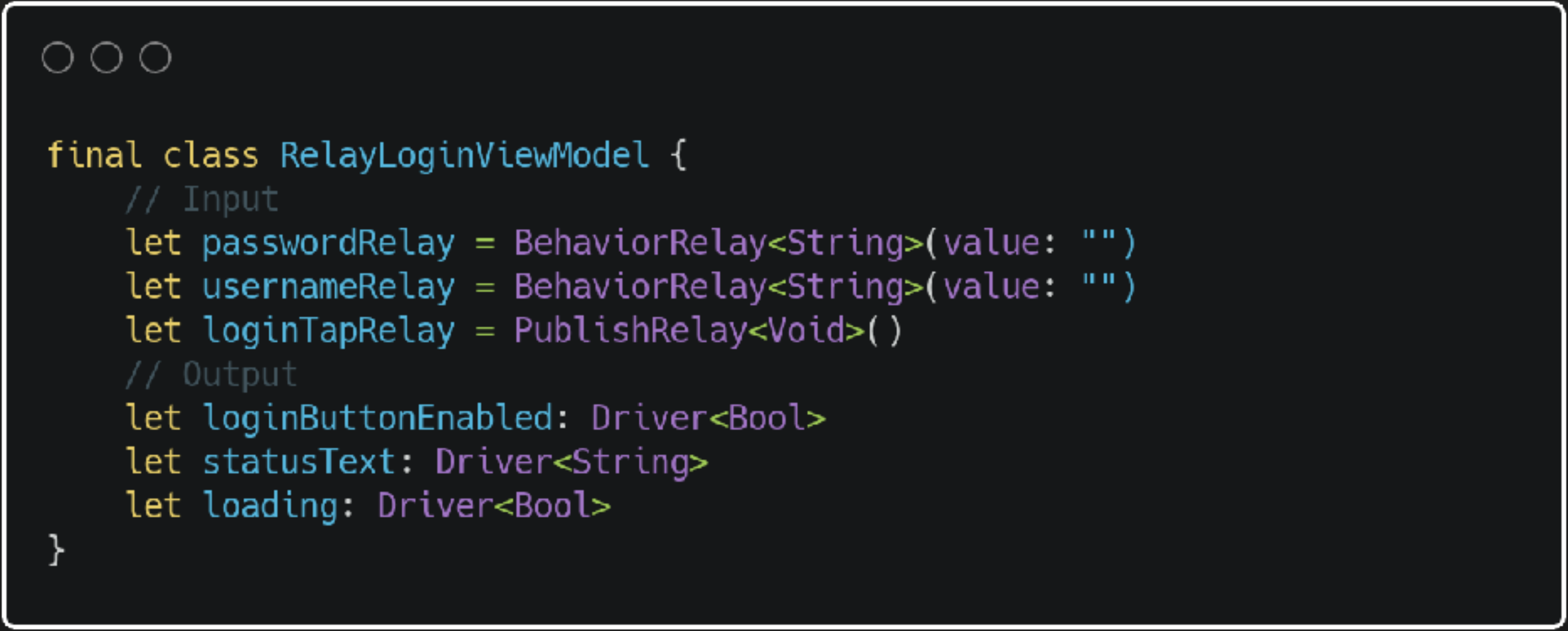
```swift
protocol KSLoginViewModelInputs {
    func passwordTextDidChange(_ text: String)
    func userNameTextDidChange(_ text: String)
    func loginButtonPressed()
}

protocol KSLoginViewModelOutputs {
    var loginButtonEnabled: Driver<Bool> { get }
    var statusText: Driver<String> { get }
    var loading: Driver<Bool> { get }
}

protocol KSLoginViewModelType {
    var inputs: KSLoginViewModelInputs { get }
    var outputs: KSLoginViewModelOutputs { get }
}
```

INPUT/OUTPUT STYLE



```swift
final class RxLoginViewModel {
    struct Input {
        let username: Signal<String>
        let password: Signal<String>
        let loginTap: Signal<Void>
    }

    struct Output {
        let loginButtonEnabled: Driver<Bool>
        let statusText: Driver<String>
        let loading: Driver<Bool>
    }

    func transform(input: Input) -> Output
}
```

USING VARIABLES/SUBJECTS/RELAYS



```swift
final class RelayLoginViewModel {
    // Input
    let passwordRelay = BehaviorRelay<String>(value: "")
    let usernameRelay = BehaviorRelay<String>(value: "")
    let loginTapRelay = PublishRelay<Void>()
    // Output
    let loginButtonEnabled: Driver<Bool>
    let statusText: Driver<String>
    let loading: Driver<Bool>
}
```

ISSUES OF THESE APPROACHES:

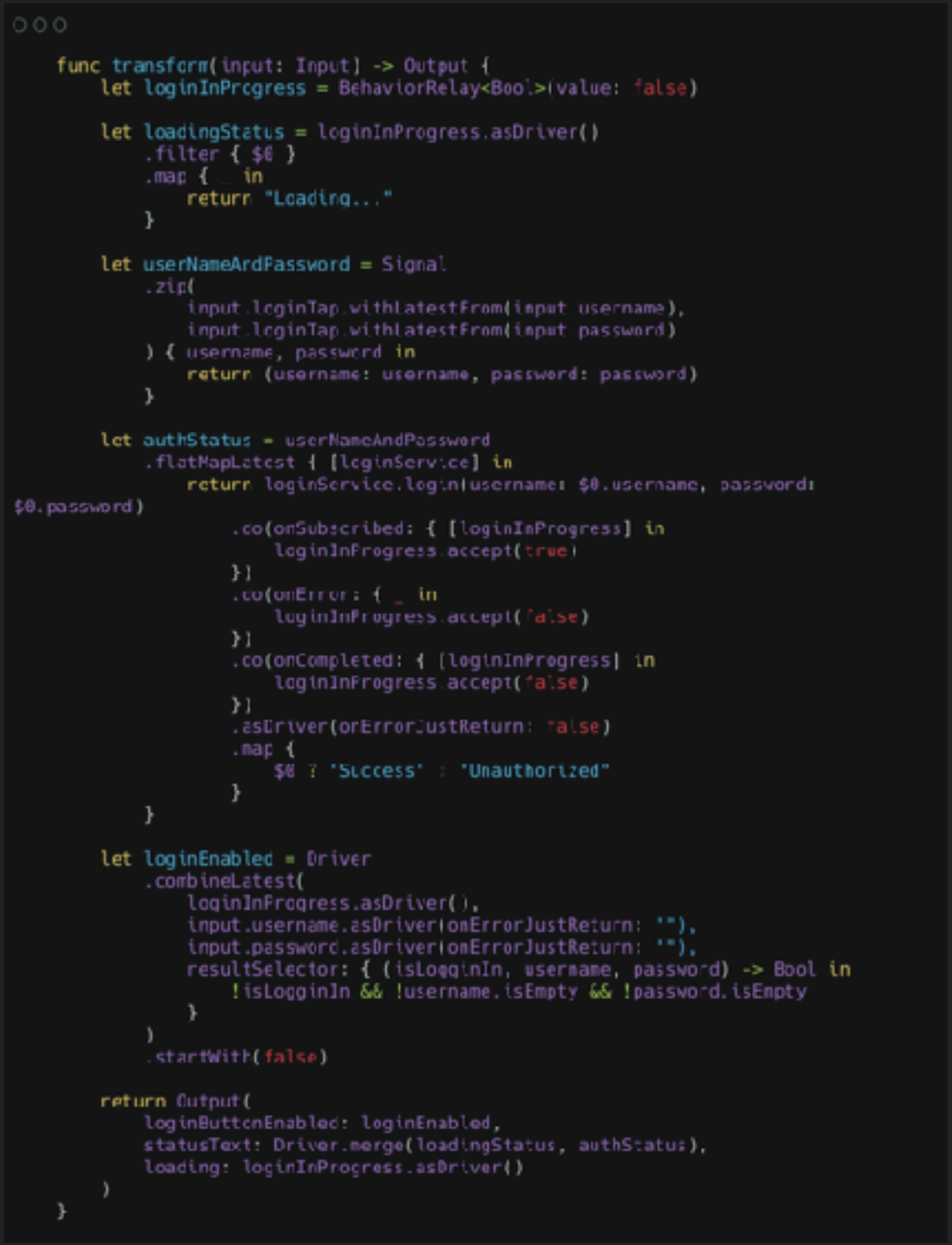
ISSUES OF THESE APPROACHES:

- Requires advanced knowledge

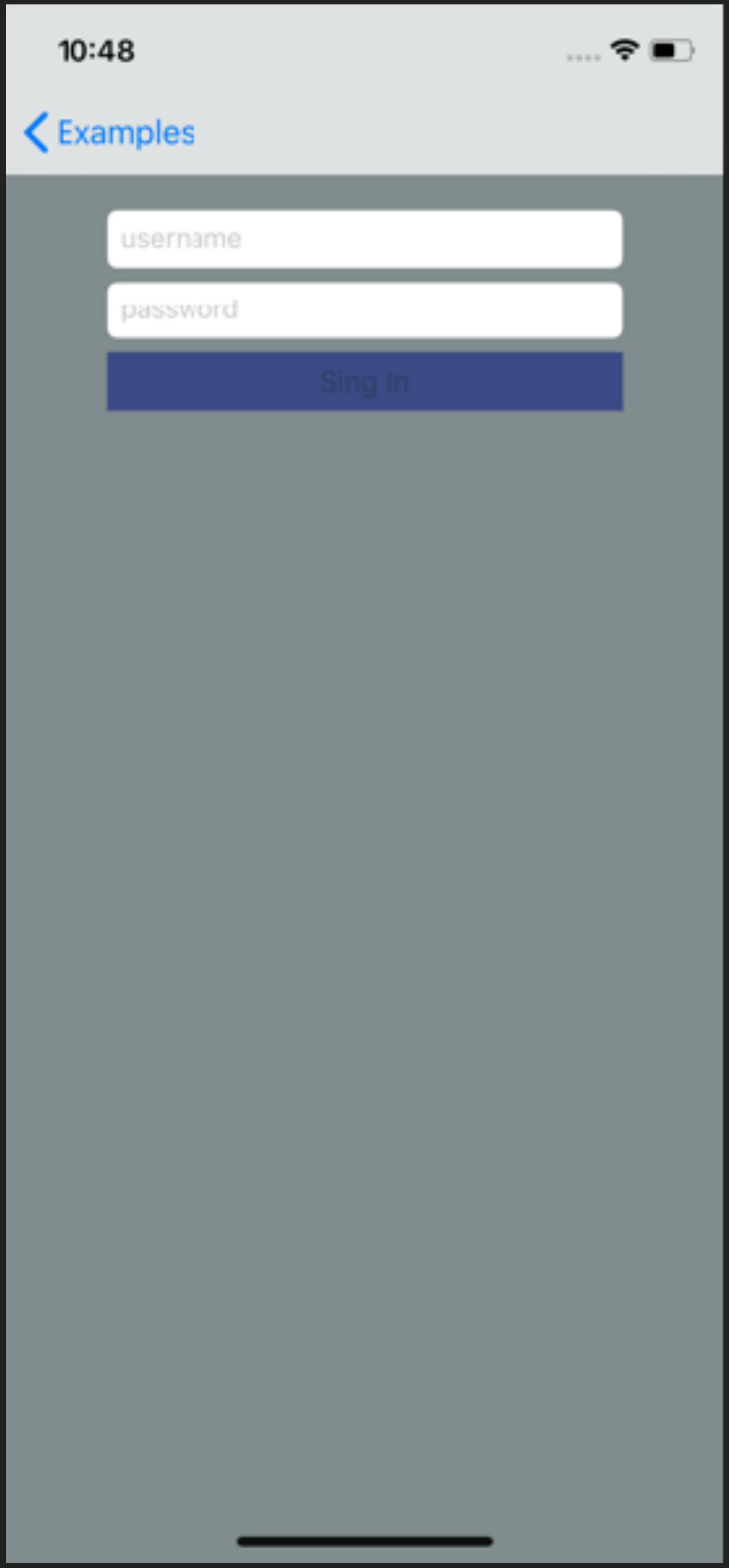
ISSUES OF THESE APPROACHES:

- Requires advanced knowledge
- It can get messy pretty fast

ISSUES OF THESE APPROACHES:



```
    func transform(input: Input) -> Output {
        let loginInProgress = BehaviorRelay<Bool>(value: false)

        let loadingStatus = loginInProgress.asDriver()
            .filter { $0 }
            .map { _ in
                return "Loading..."
            }

        let userNameAndPassword = Signal
            .zip(
                input.loginTap.withLatestFrom(input.username),
                input.loginTap.withLatestFrom(input.password)
            ) { username, password in
                return (username: username, password: password)
            }

        let authStatus = userNameAndPassword
            .flatMapLatest { [loginService] in
                return loginService.login(username: $0.username, password:
$0.password)
                    .do(onSubscribed: { [loginInProgress] in
                        loginInProgress.accept(true)
                    })
                    .do(onError: { _ in
                        loginInProgress.accept(false)
                    })
                    .do(onCompleted: { [loginInProgress] in
                        loginInProgress.accept(false)
                    })
                    .asDriver(onErrorJustReturn: false)
                    .map {
                        $0 ? "Success" : "Unauthorized"
                    }
            }

        let loginEnabled = Driver
            .combineLatest(
                loginInProgress.asDriver(),
                input.username.asDriver(onErrorJustReturn: ""),
                input.password.asDriver(onErrorJustReturn: ""),
                resultSelector: { (isLogginIn, username, password) -> Bool in
                    !isLogginIn && !username.isEmpty && !password.isEmpty
                }
            )
            .startWith(false)

        return Output(
            loginButtonEnabled: loginEnabled,
            statusText: Driver.merge(loadingStatus, authStatus),
            loading: loginInProgress.asDriver()
        )
    }
```



ISSUES OF THESE APPROACHES:

- Requires advanced knowledge
- It can get messy pretty fast
- Hard to debug

ISSUES OF THESE APPROACHES:

- Requires advanced knowledge
- It can get messy pretty fast
- Hard to debug
- No way to get a snapshot of a current “State”

ISSUES OF THESE APPROACHES:

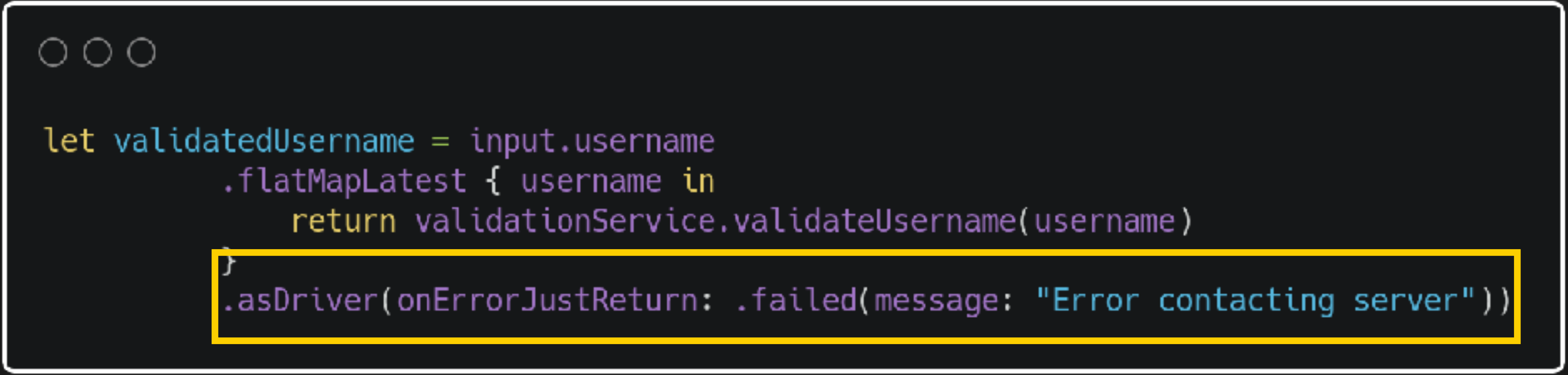
- Requires advanced knowledge
- It can get messy pretty fast
- Hard to debug
- No way to get a snapshot of a current “State”
- Provides only interface guidelines

ISSUES OF THESE APPROACHES:

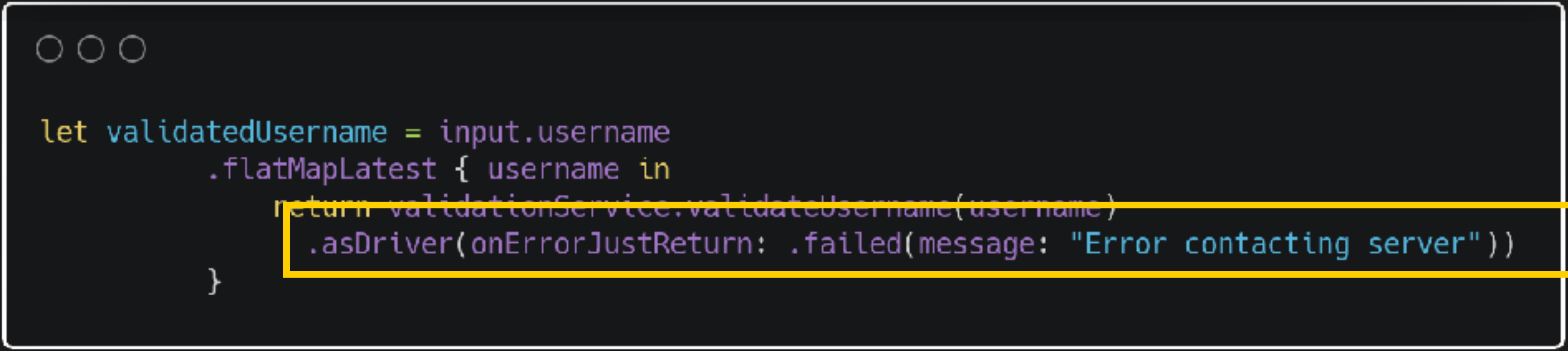
- Requires advanced knowledge
- It can get messy pretty fast
- Hard to debug
- No way to get a snapshot of a current “State”
- Provides only interface guidelines
- Error prone

ERROR PRONE

CATCHING ERRORS

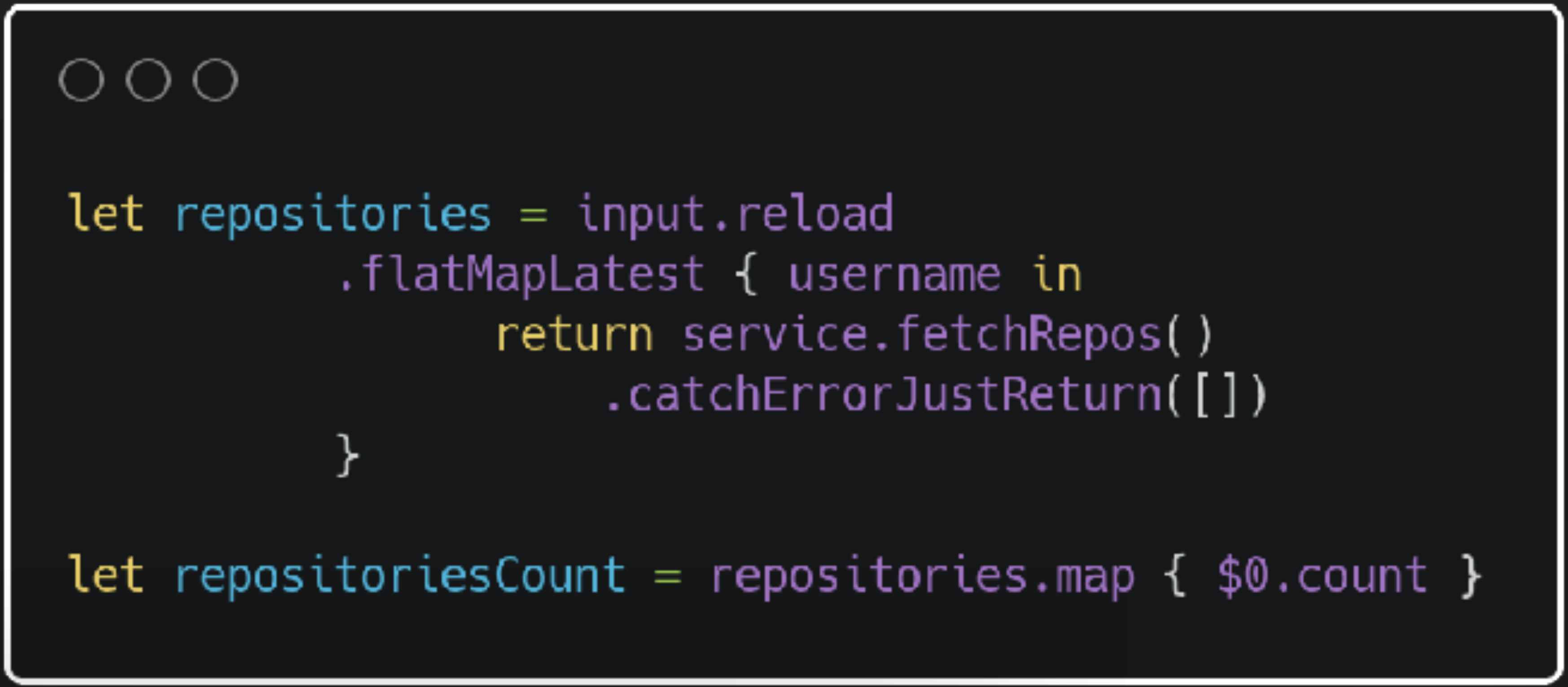


```
let validatedUsername = input.username
          .flatMapLatest { username in
              return validationService.validateUsername(username)
          }
          .asDriver(onErrorJustReturn: .failed(message: "Error contacting server"))
```

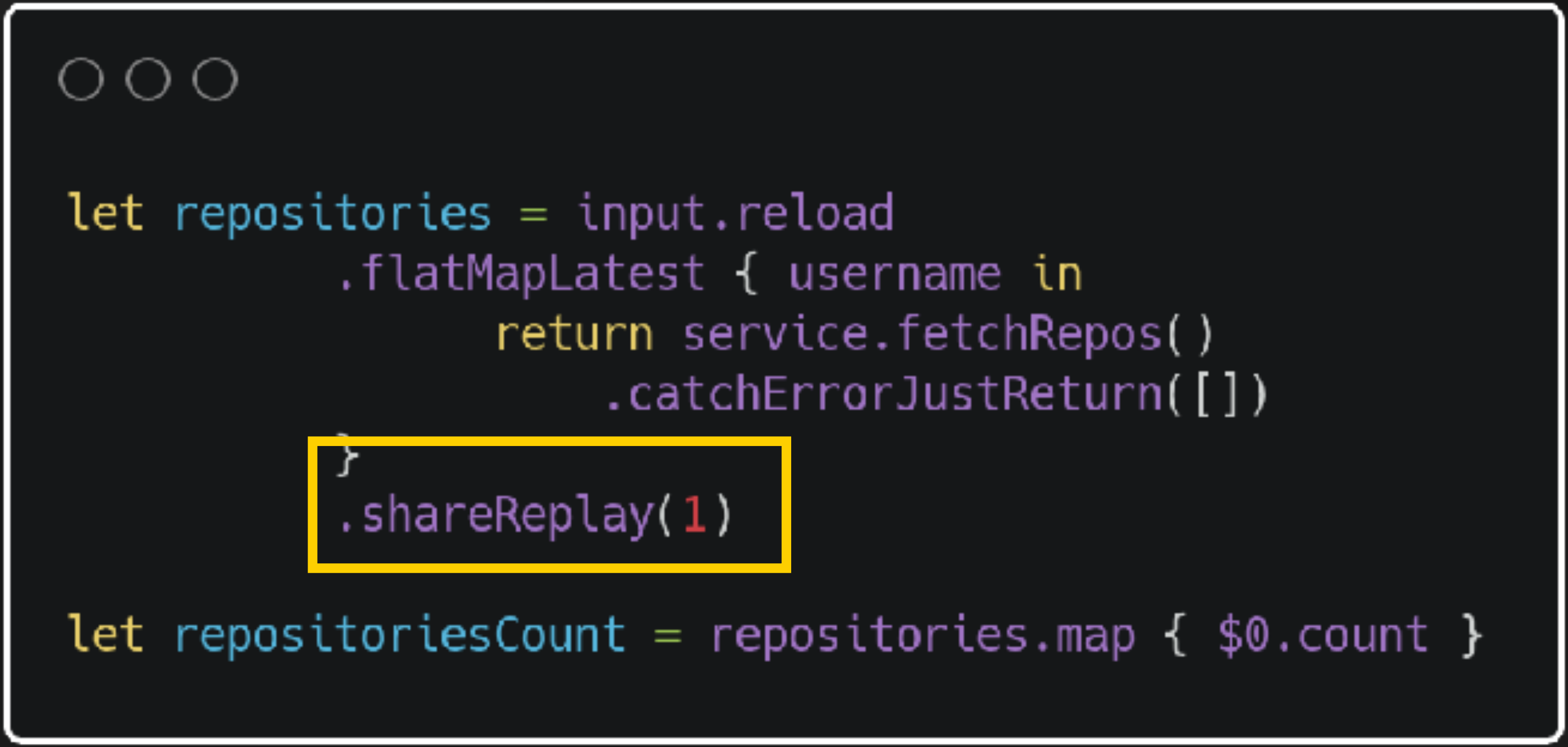


```
let validatedUsername = input.username
          .flatMapLatest { username in
              ~~return validationService.validateUsername(username)~~
                .asDriver(onErrorJustReturn: .failed(message: "Error contacting server"))
          }
```

SHARING IS CARING

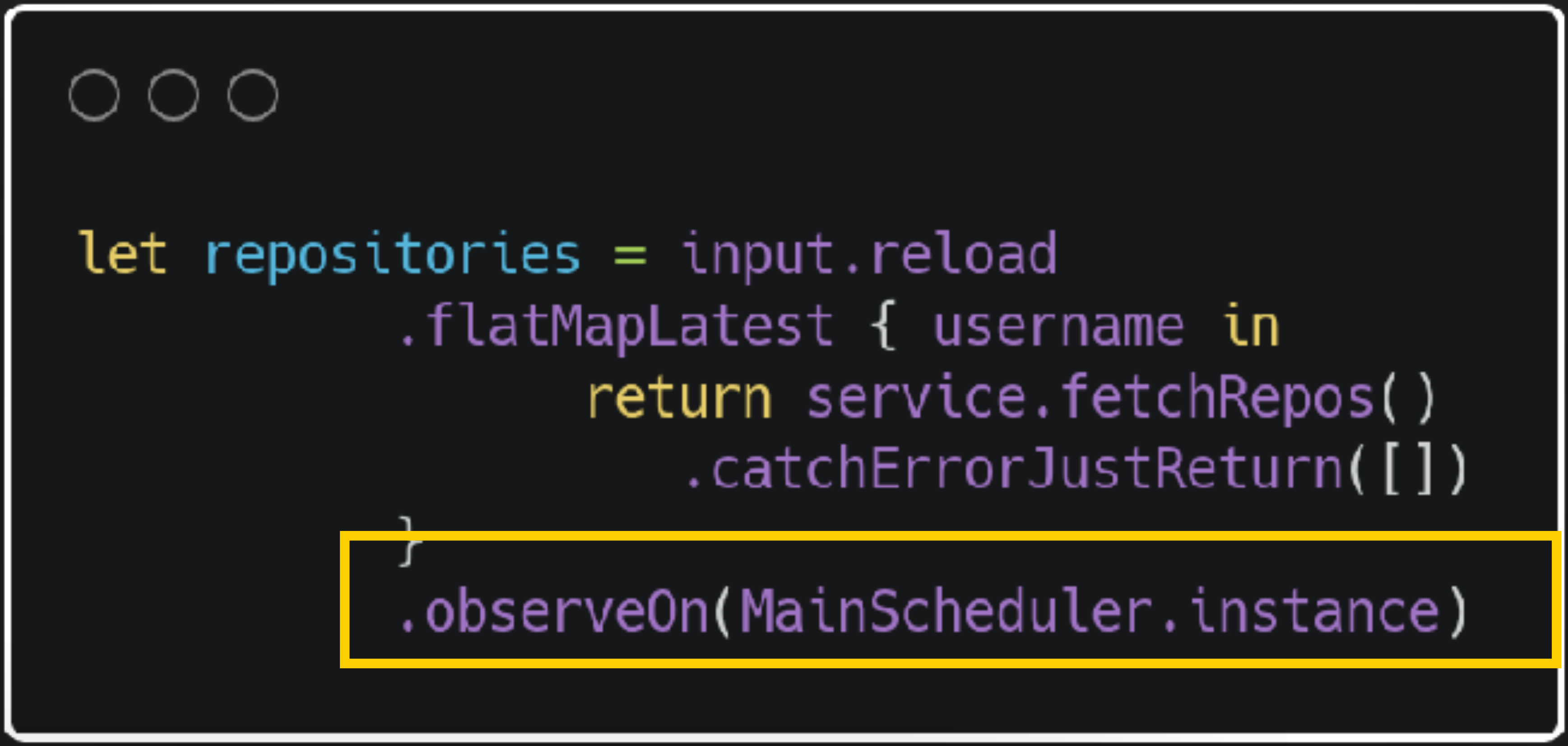


```
let repositories = input.reload
          .flatMapLatest { username in
                return service.fetchRepos()
                    .catchErrorJustReturn([])
          }

let repositoriesCount = repositories.map { $0.count }
```

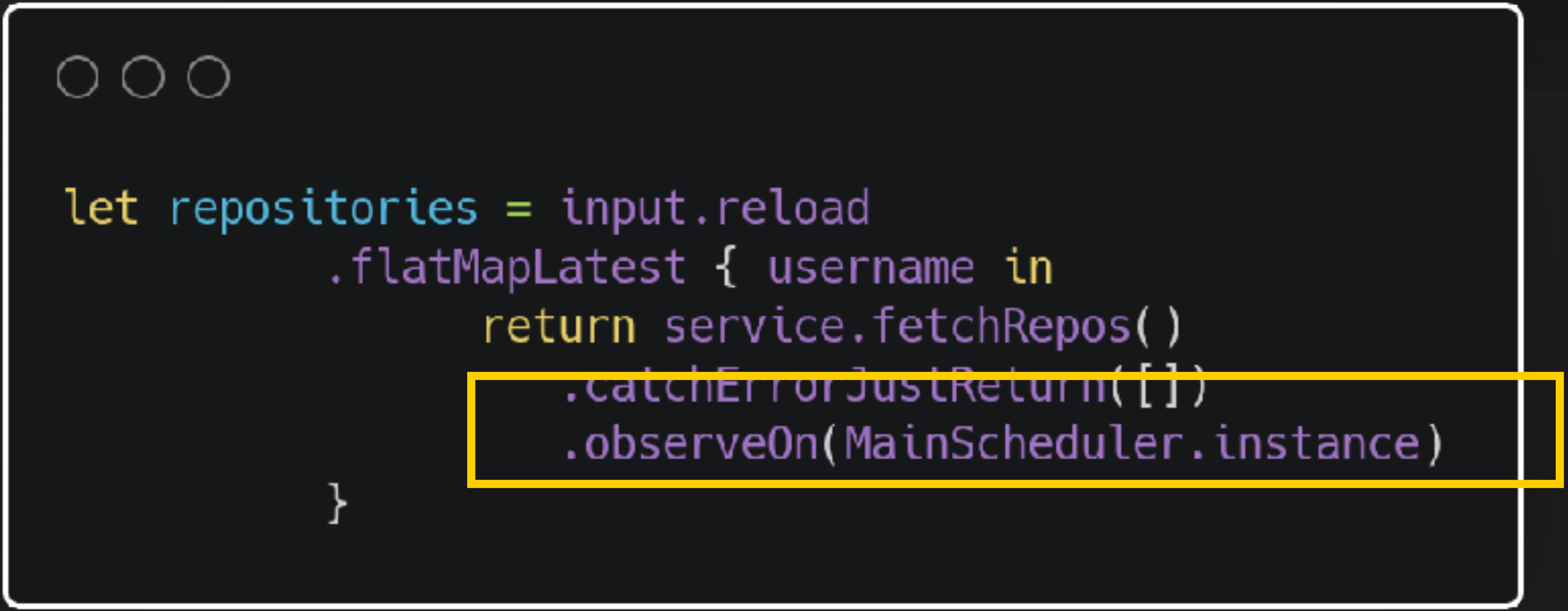


```
let repositories = input.reload
          .flatMapLatest { username in
                return service.fetchRepos()
                    .catchErrorJustReturn([])
          }
          .shareReplay(1)

let repositoriesCount = repositories.map { $0.count }
```

CANCELATION



```
let repositories = input.reload
            .flatMapLatest { username in
                    return service.fetchRepos()
                        .catchErrorJustReturn([])
            }
            .observeOn(MainScheduler.instance)
```



```
let repositories = input.reload
            .flatMapLatest { username in
                    return service.fetchRepos()
                        .catchErrorJustReturn([])
                        .observeOn(MainScheduler.instance)
            }
```

GOALS

GOALS

- Common approach

GOALS

- Common approach
- Reduce/remove common mistakes

GOALS

- Common approach
- Reduce/remove common mistakes
- Not require advanced knowledge of FRP

IF YOU WANT TO MAKE PEOPLE DO SOMETHING, YOU NEED TO MAKE IT EASIER FOR THEM.

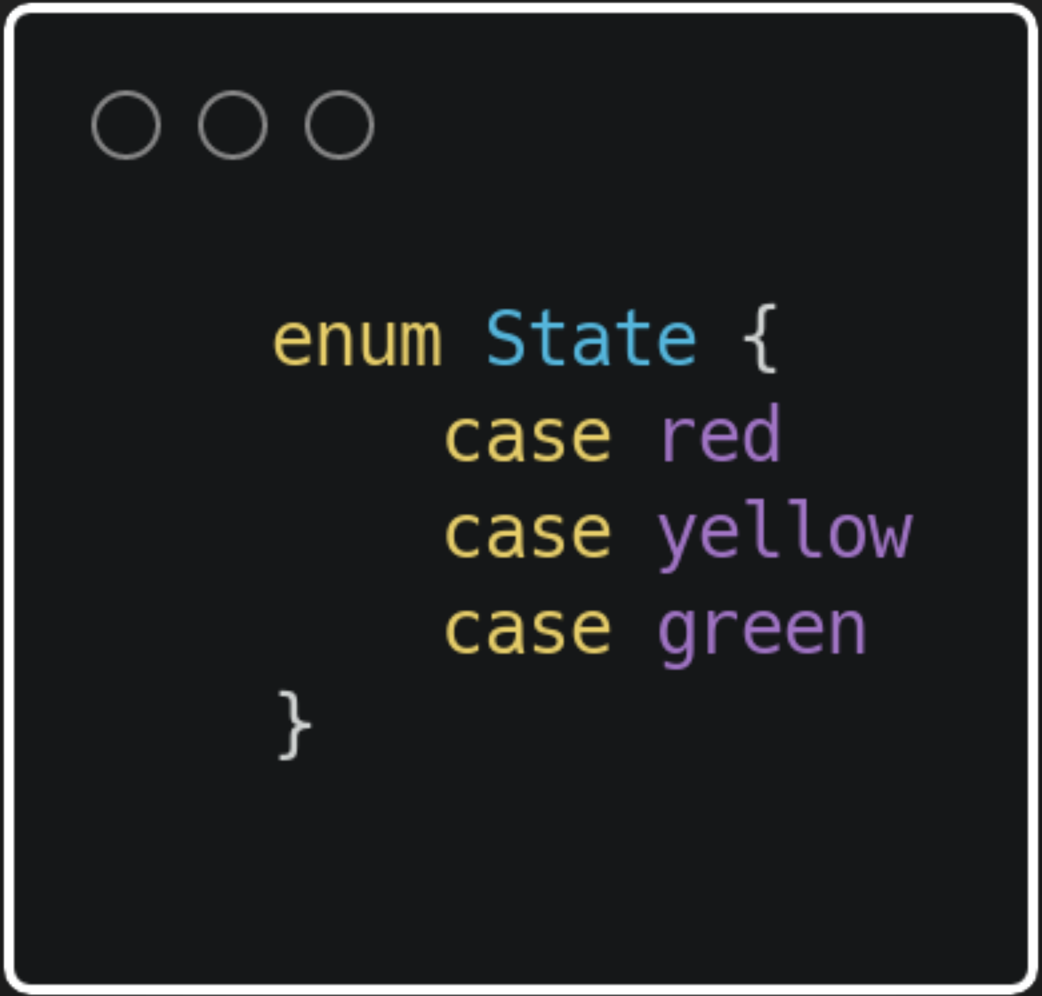
REACTIVE FEEDBACK

STATE

EFFECTS

EVENT

STATE



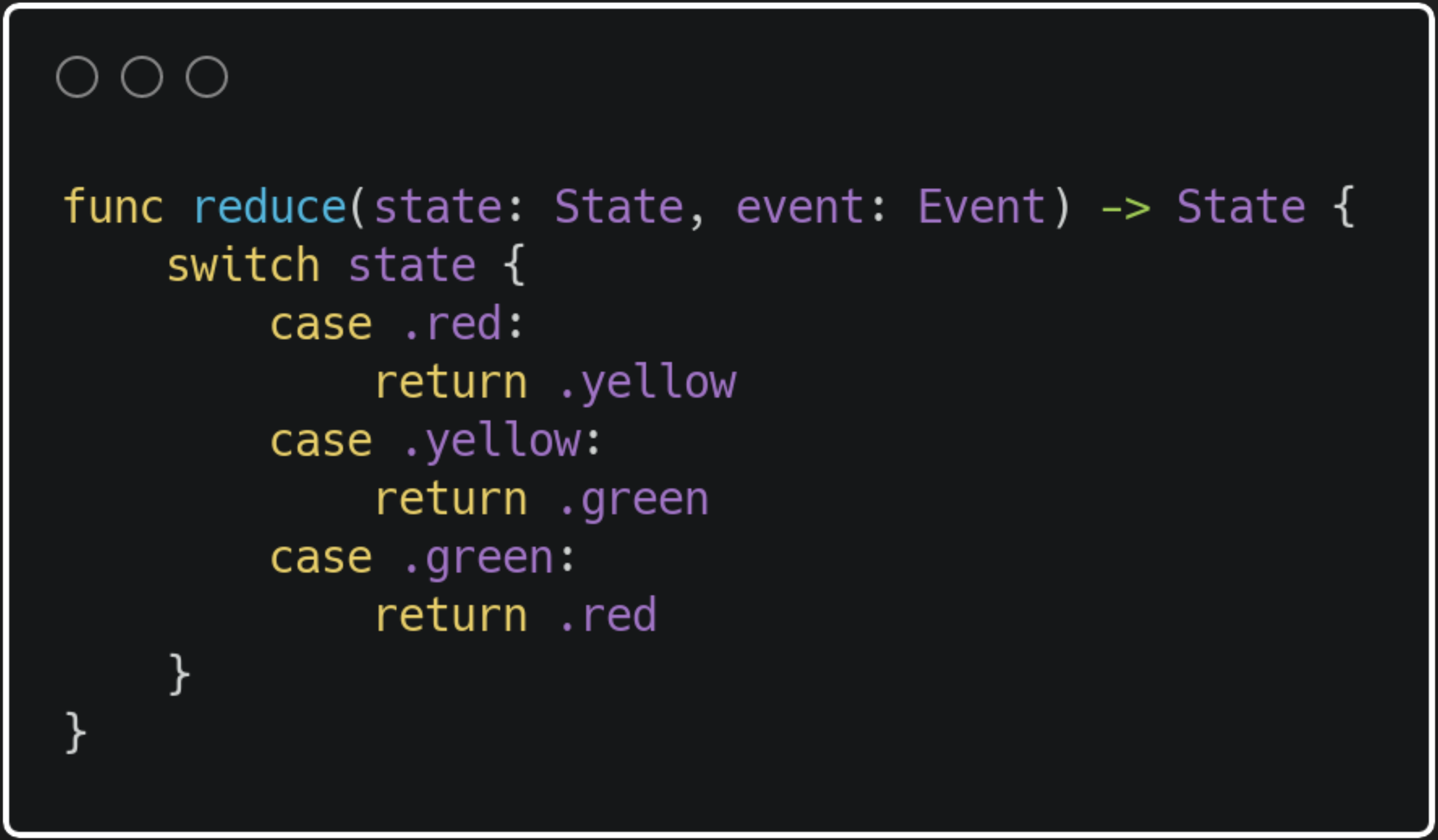
```
enum State {
    case red
    case yellow
    case green
}
```

EVENT

```
enum Event {
    case next
}
```

STATE + EVENT = STATE

REDUCER



```swift
func reduce(state: State, event: Event) -> State {
    switch state {
        case .red:
            return .yellow
        case .yellow:
            return .green
        case .green:
            return .red
    }
}
```

EFFECTS

EFFECTS

- Timers

EFFECTS

- Timers
- Network requests

EFFECTS

- Timers
- Network requests
- DB queries

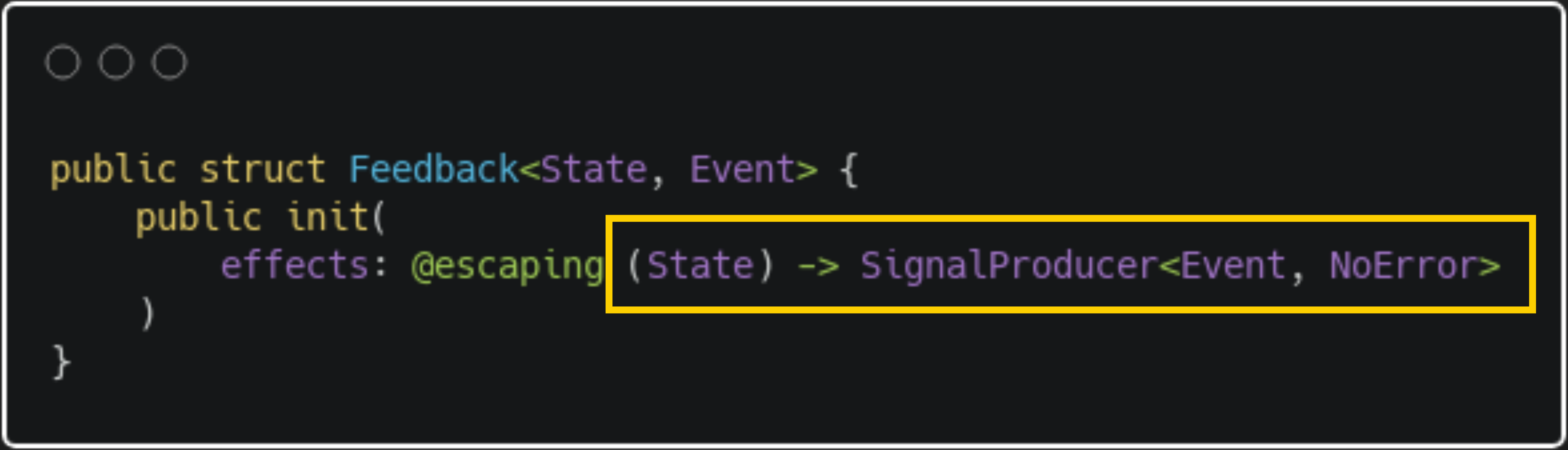
EFFECTS

- Timers
- Network requests
- DB queries
- Bluetooth connections

EFFECTS

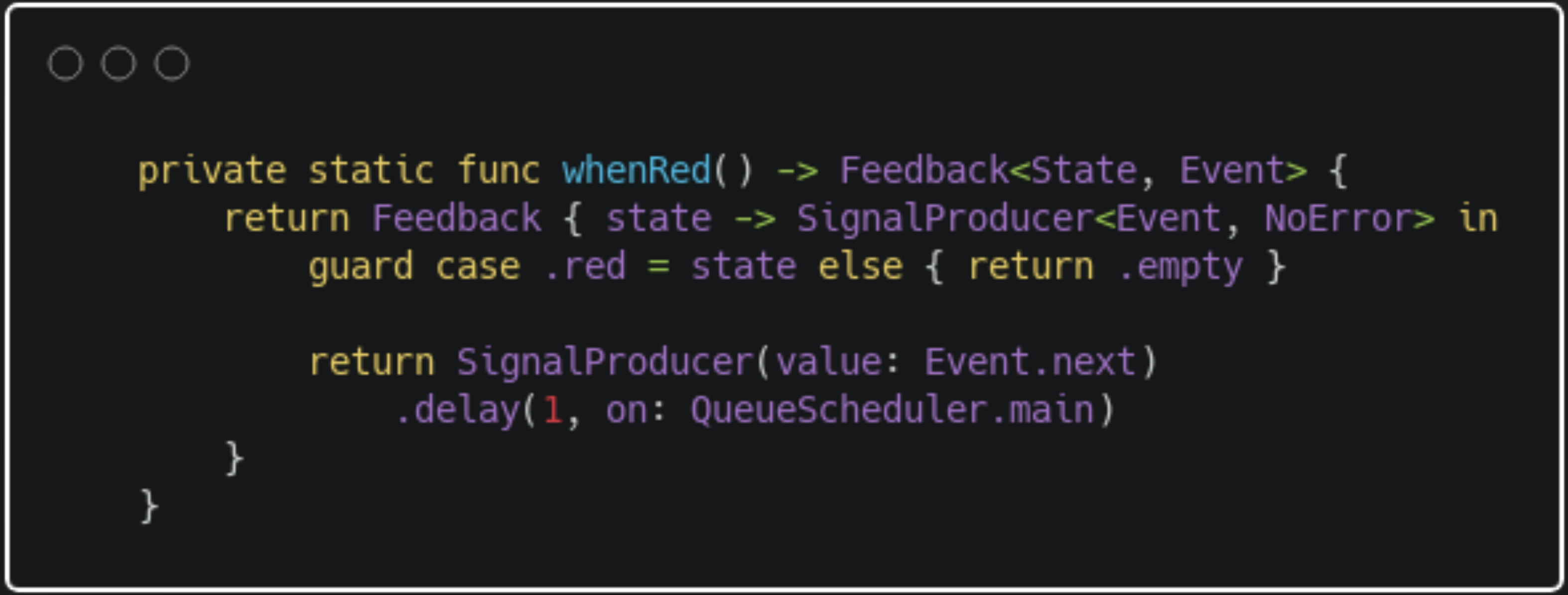
- Timers
- Network requests
- DB queries
- Bluetooth connections
- User input

FEEDBACK



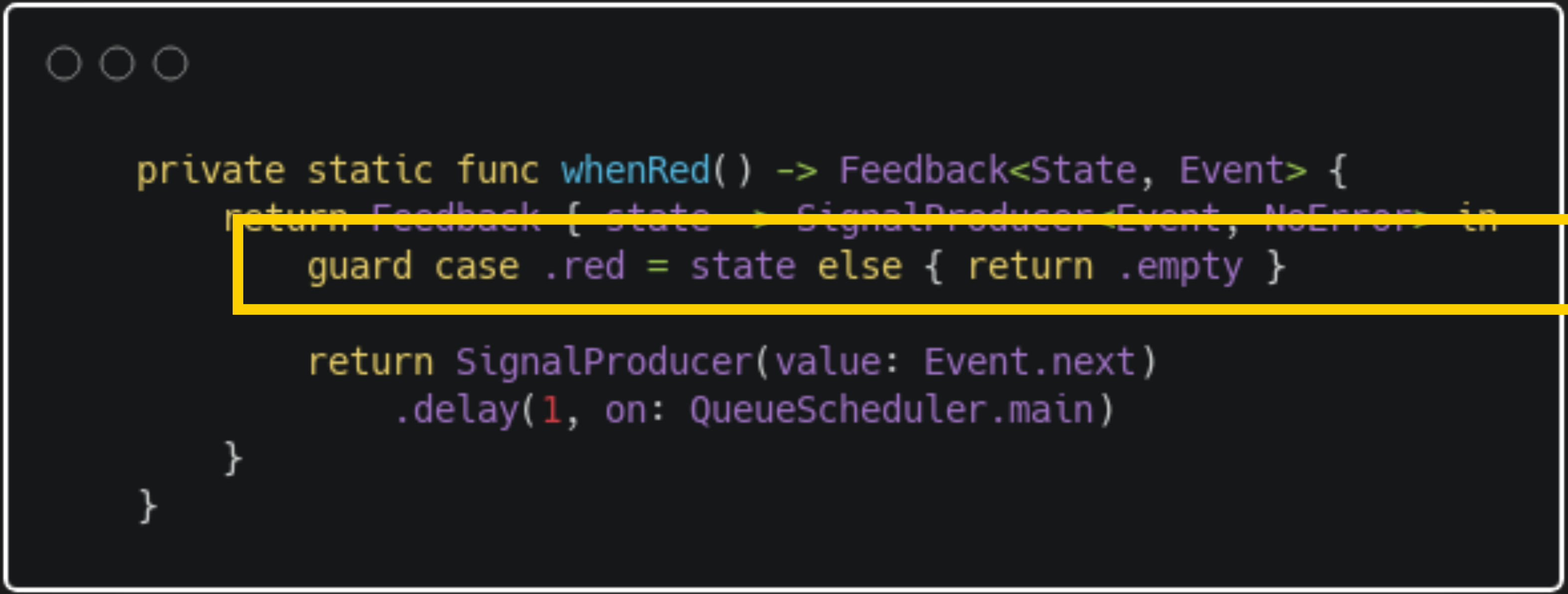
```
public struct Feedback<State, Event> {
    public init(
        effects: @escaping (State) -> SignalProducer<Event, NoError>
    )
}
```

FEEDBACK



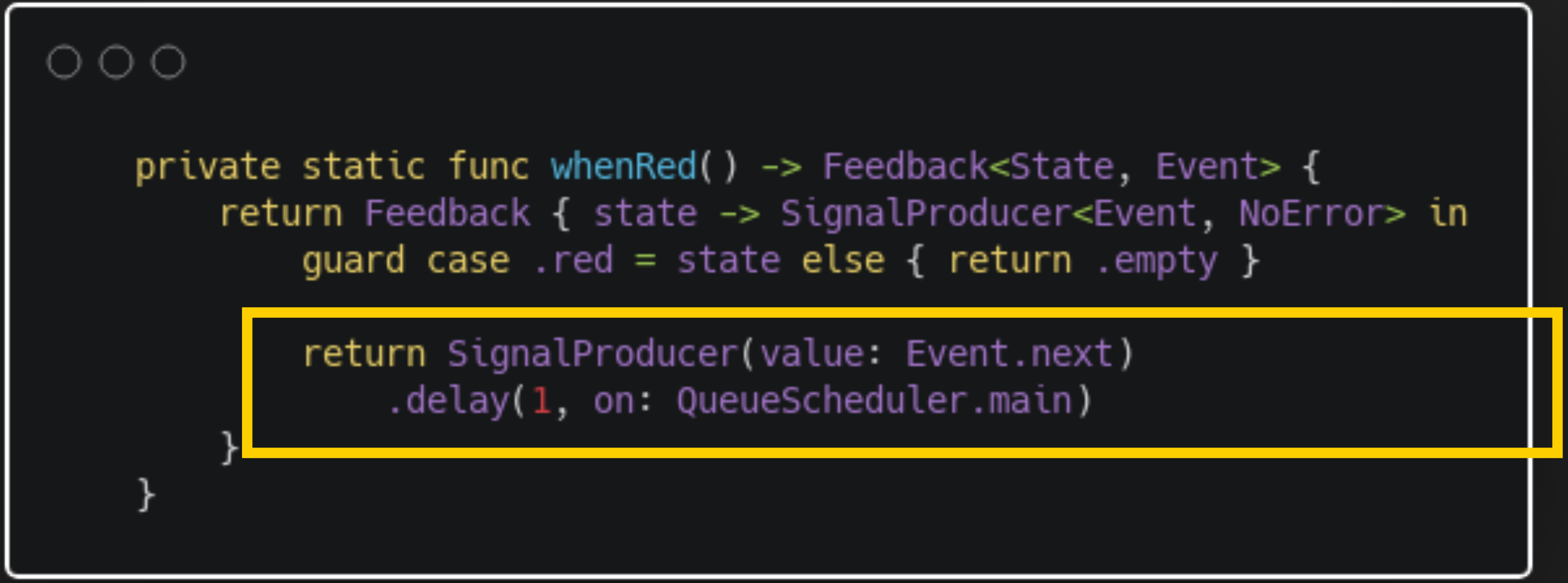
```
private static func whenRed() -> Feedback<State, Event> {
    return Feedback { state -> SignalProducer<Event, NoError> in
        guard case .red = state else { return .empty }

        return SignalProducer(value: Event.next)
            .delay(1, on: QueueScheduler.main)
    }
}
```

FEEDBACK



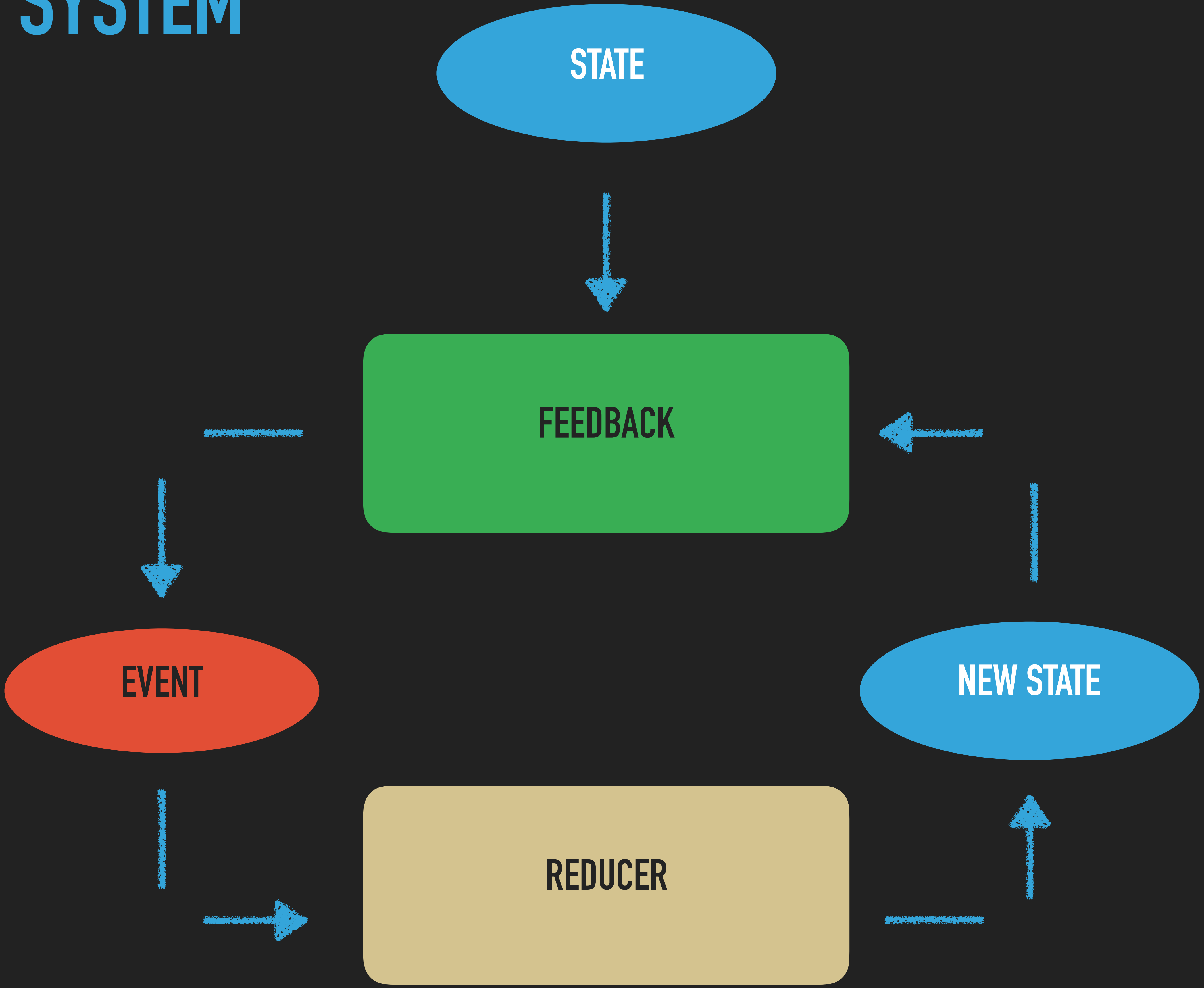
```
private static func whenRed() -> Feedback<State, Event> {
    return Feedback { state -> SignalProducer<Event, NoError> in
        guard case .red = state else { return .empty }

        return SignalProducer(value: Event.next)
            .delay(1, on: QueueScheduler.main)
    }
}
```

FEEDBACK

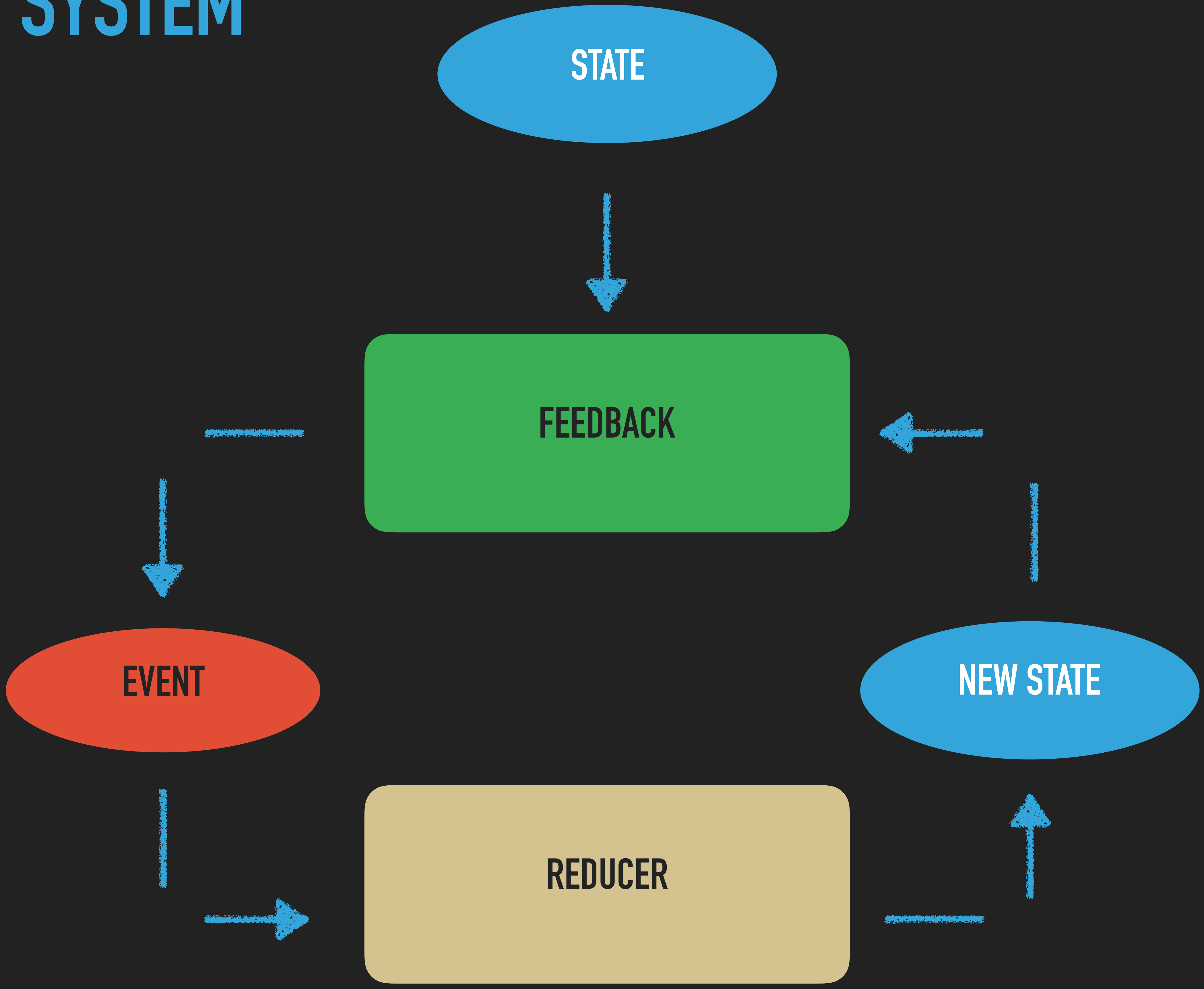


```
private static func whenRed() -> Feedback<State, Event> {
    return Feedback { state -> SignalProducer<Event, NoError> in
        guard case .red = state else { return .empty }

        return SignalProducer(value: Event.next)
            .delay(1, on: QueueScheduler.main)
    }
}
```

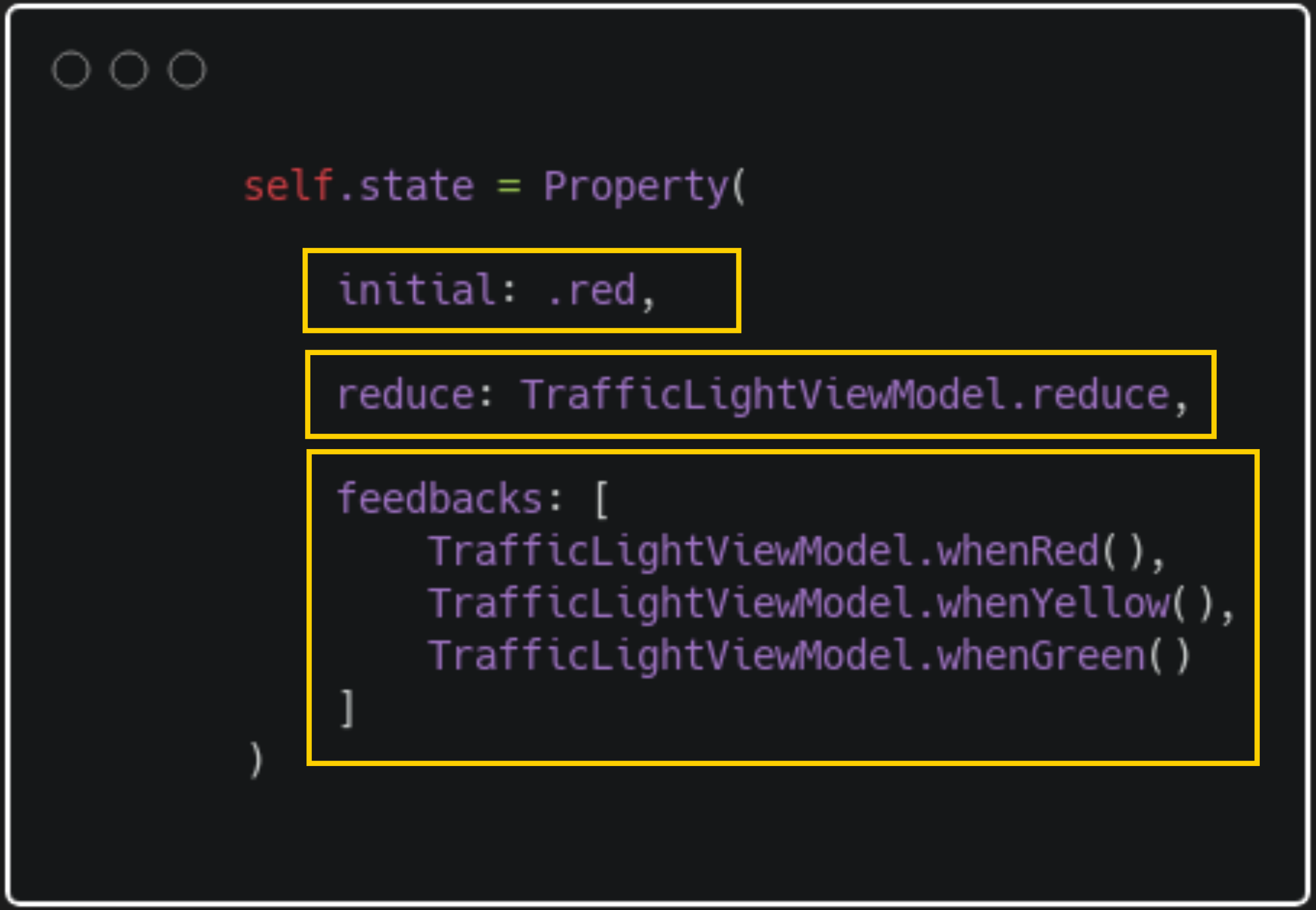
SYSTEM



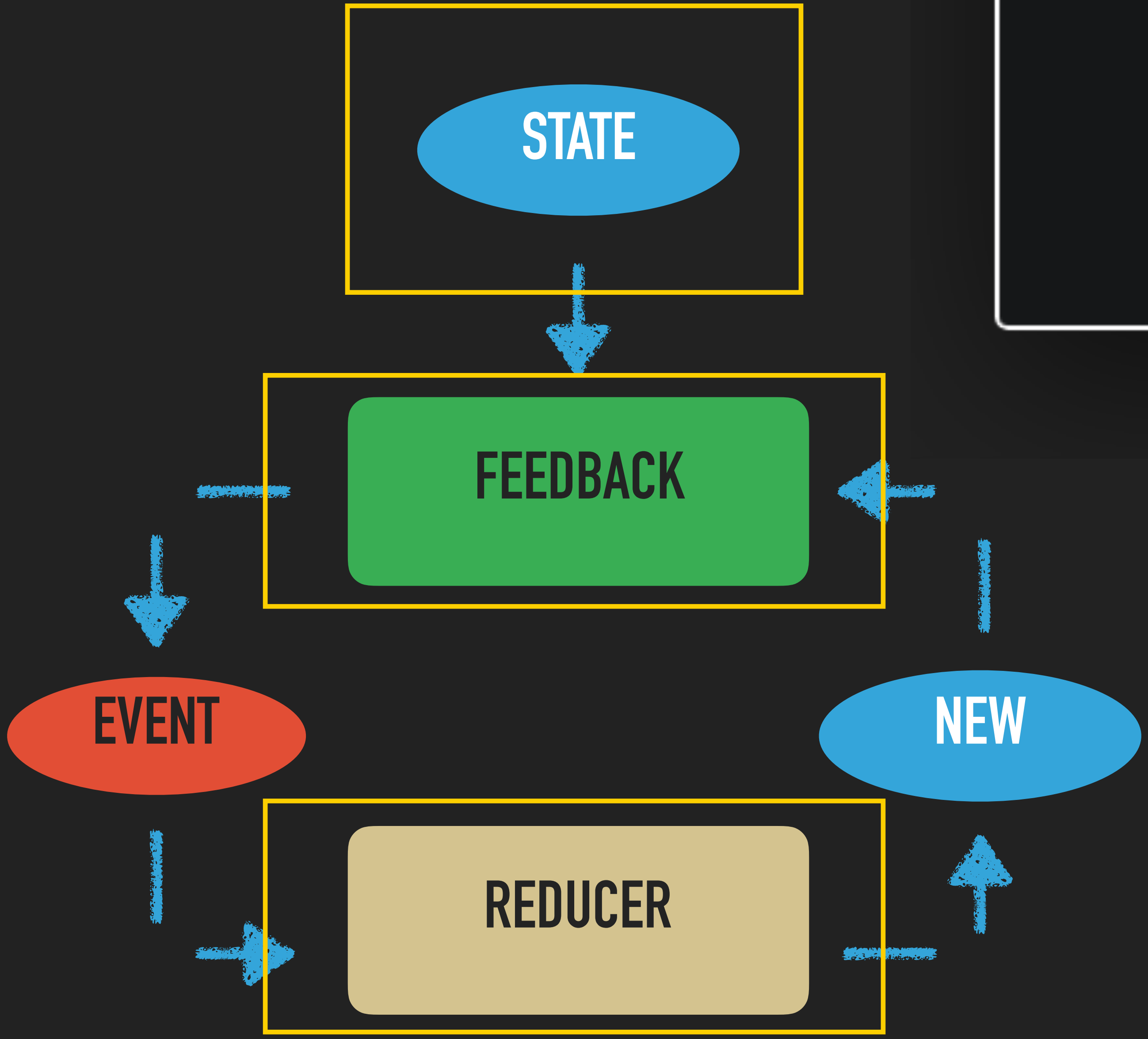
SYSTEM



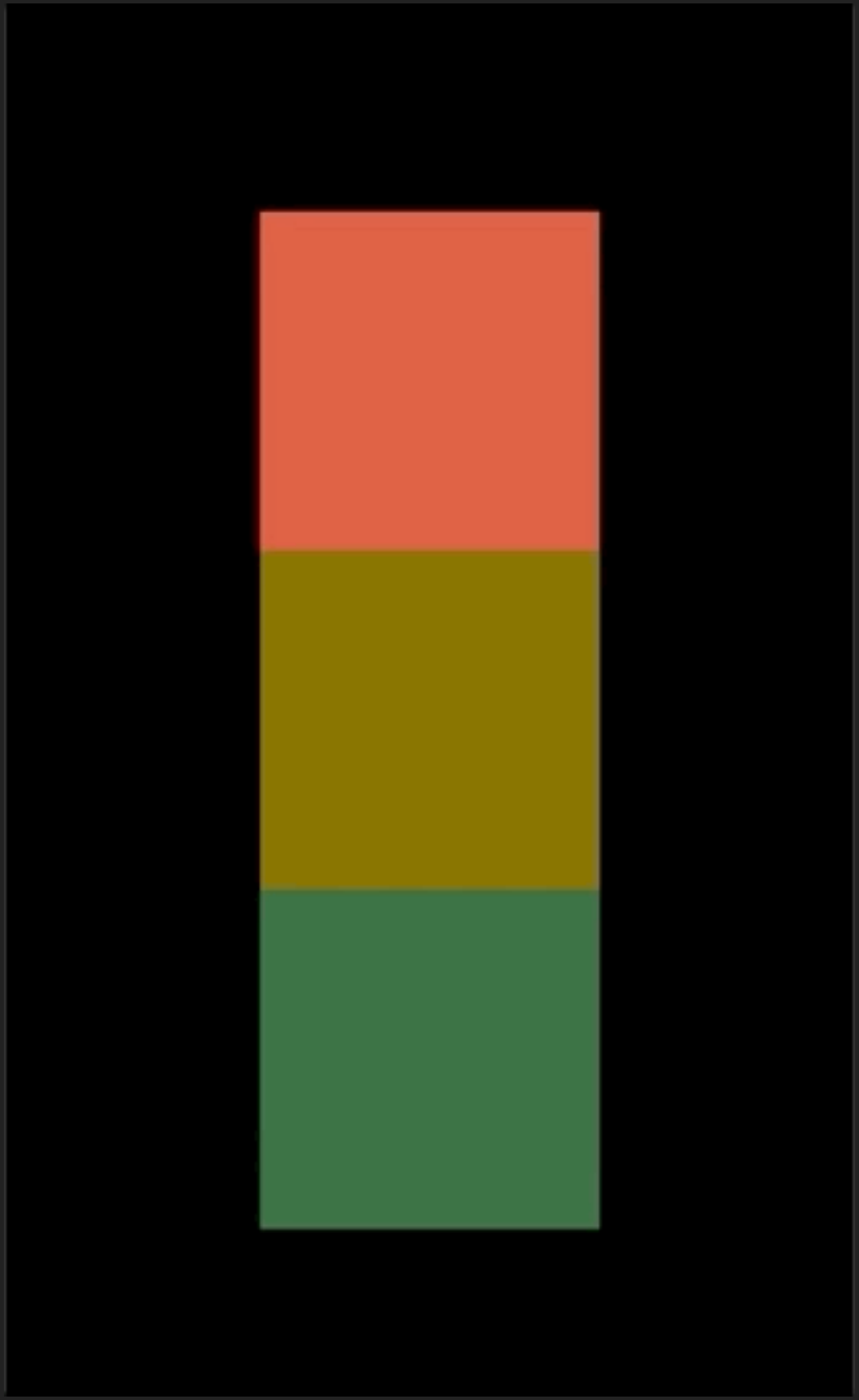
SYSTEM



```
self.state = Property(
    initial: .red,
    reduce: TrafficLightViewModel.reduce,
    feedbacks: [
        TrafficLightViewModel.whenRed(),
        TrafficLightViewModel.whenYellow(),
        TrafficLightViewModel.whenGreen()
    ]
)
```



SYSTEM



ADVANTAGES

ADVANTAGES

- Clear separation of concerns

ADVANTAGES

- Clear separation of concerns
- Easy to debug

ADVANTAGES

- Clear separation of concerns
- Easy to debug
- Easier to review

ADVANTAGES

- Clear separation of concerns
- Easy to debug
- Easier to review
- Easier to test

Q&A

@SERGDORT

LINKS

- https://github.com/sergdort/ReactiveFeedbackExamples
- https://github.com/Babylonpartners/ReactiveFeedback
- https://github.com/NoTests/RxFeedback.swift

@SERGDORT