Leveraging the power of

State Machines in Swift

Frank Courville – February 17th, 2022

Who am I?

School of Swift

- Remote workshops
- Meticulously crafted
- As short as half a day

Contact!

Don't be shy, say hi!

- @frankacy on Twitter
- @frankacy in Slack
- hello@frankcourville.com

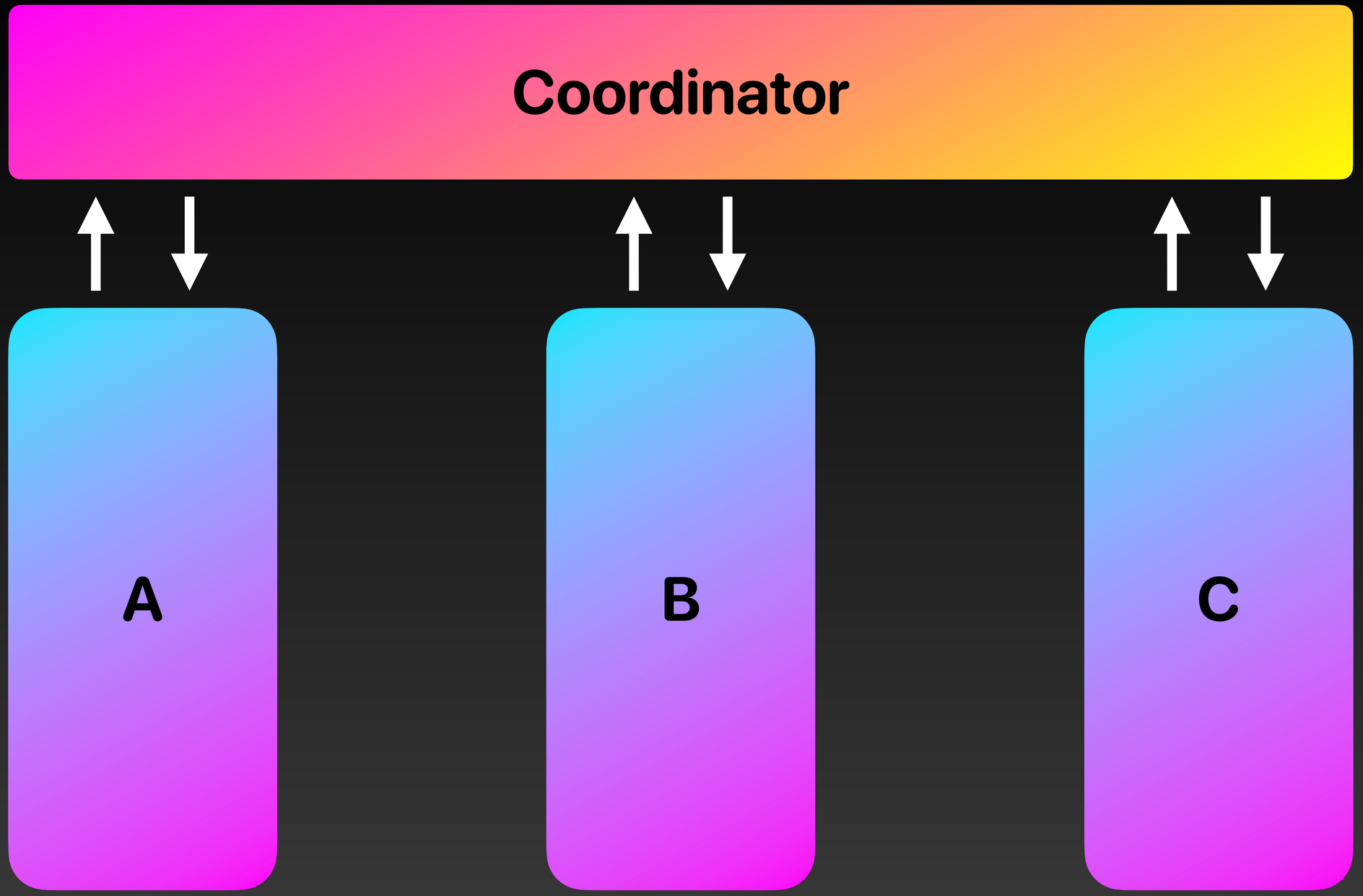
State Machines in Swift

State Machines in Swift

- How to draw state diagrams
- How to translate state diagrams into Swift
- Look at some advanced applications of state machines

"Coordinators are objects that control flow in an app."

- Soroush Khanlou


Coordinator
A
B
C

Coordinator

- How can you tell what the order of view controllers are?
- How do each of these view controllers interact with the coordinator?
- How do you test this?

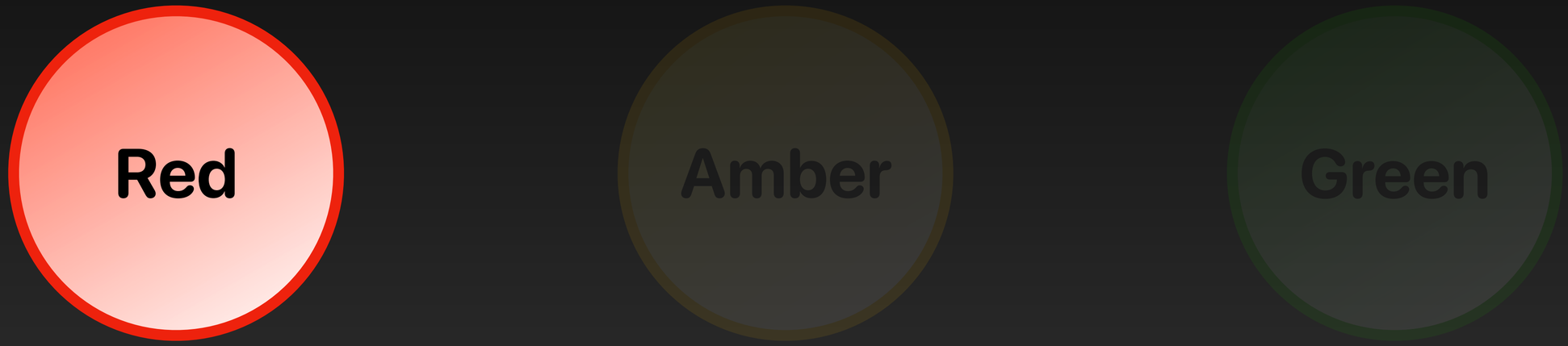
State Machines in Swift

"Pattern to represent a finite number of states, and enforce known transitions between those states"

"Pattern to represent a finite number of states, and enforce known transitions between those states"

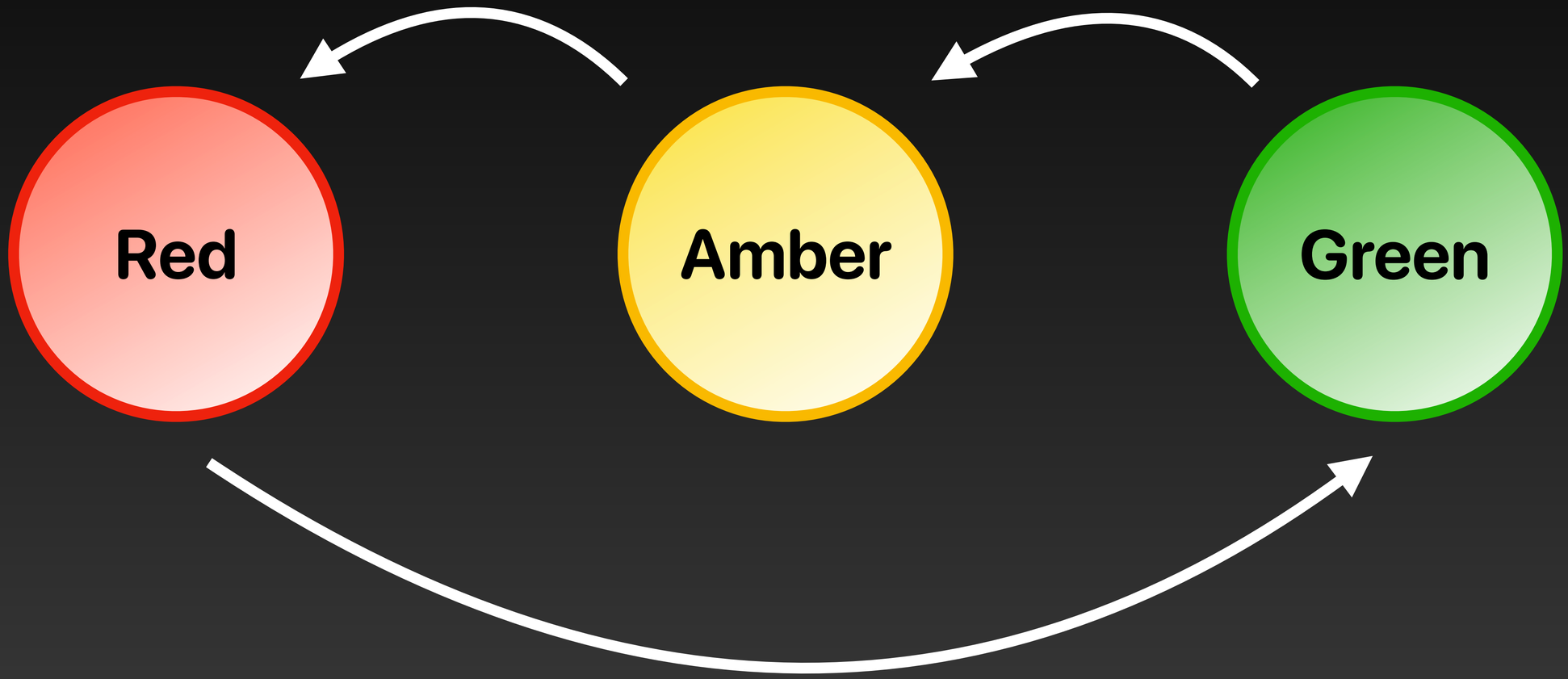
"Pattern to represent a finite number of states, and enforce known transitions between those states"

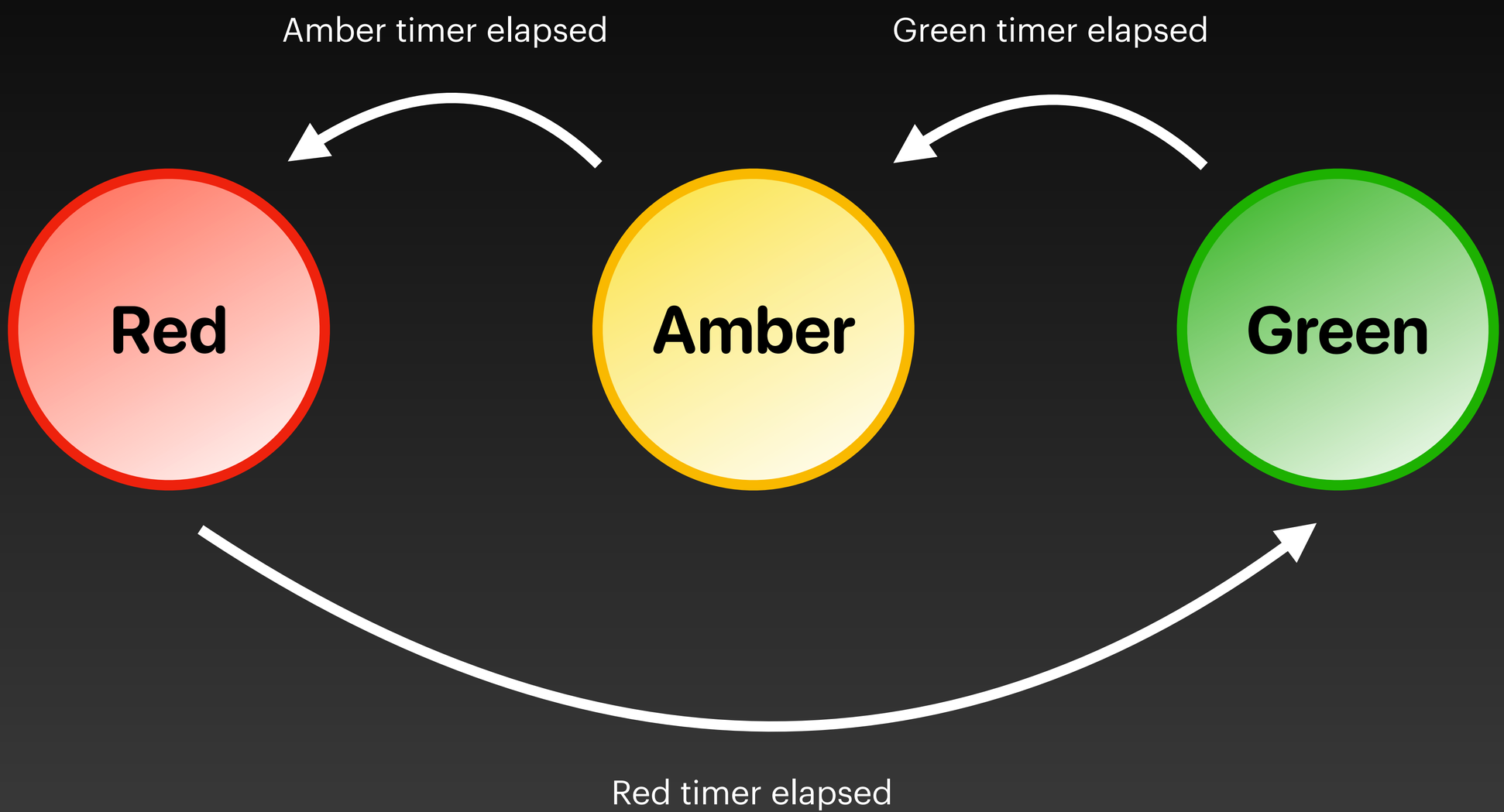
"Pattern to represent a finite number of states, and enforce known transitions between those states"


Red
Amber
Green

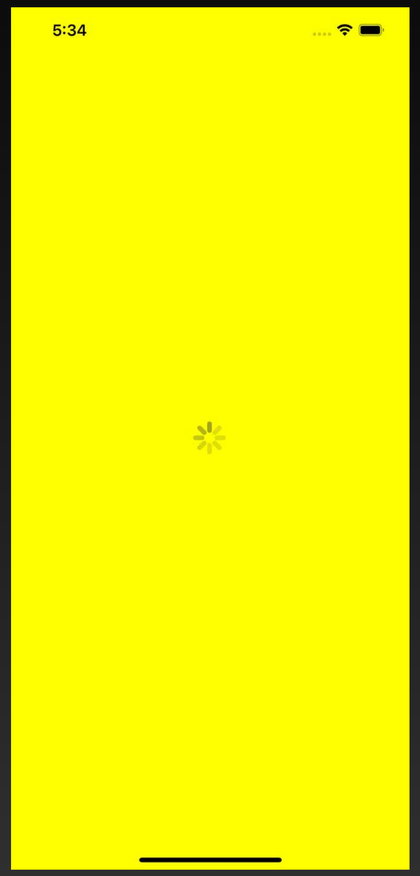

Red
Amber
Green

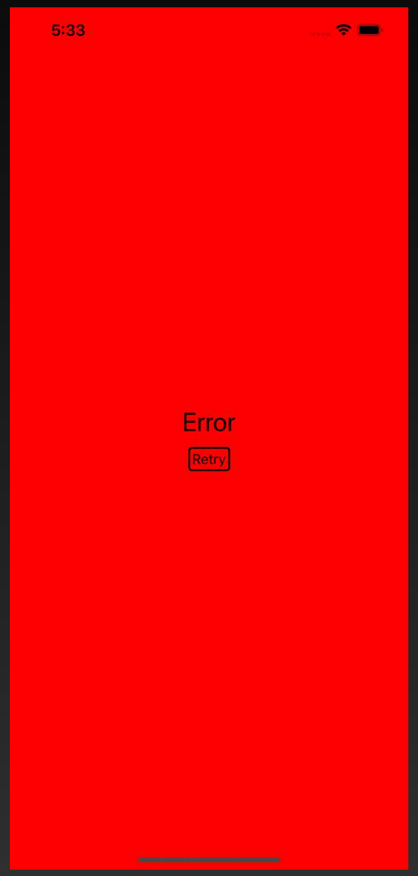

Red
Amber
Green

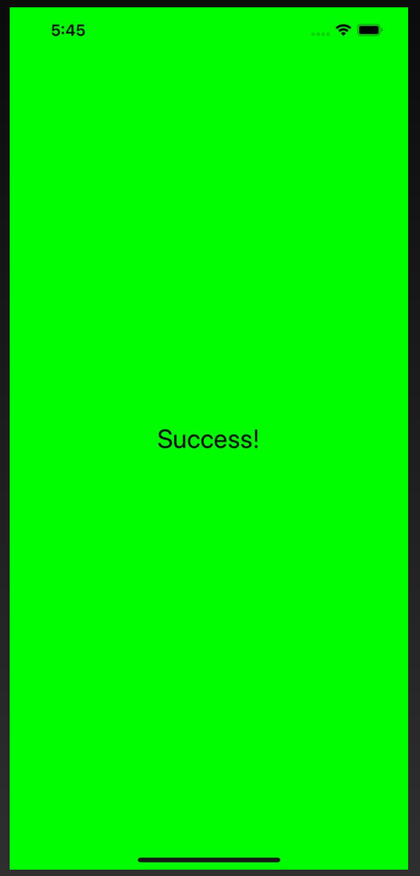

Red
Amber
Green

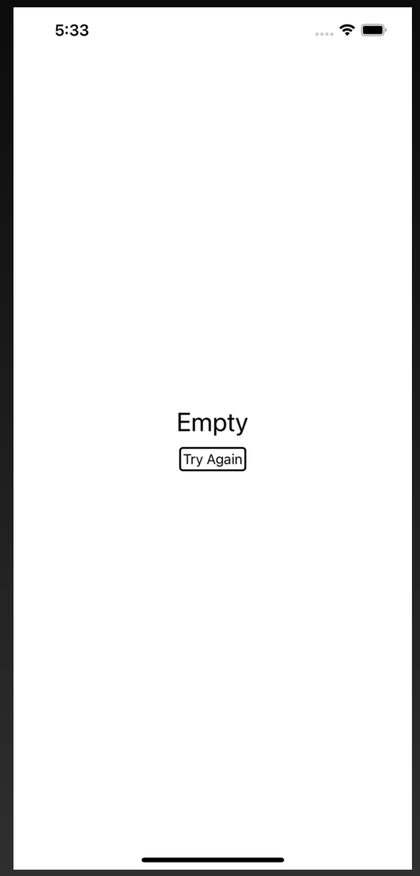

Amber timer elapsed
Green timer elapsed
Red
Amber
Green
Red timer elapsed

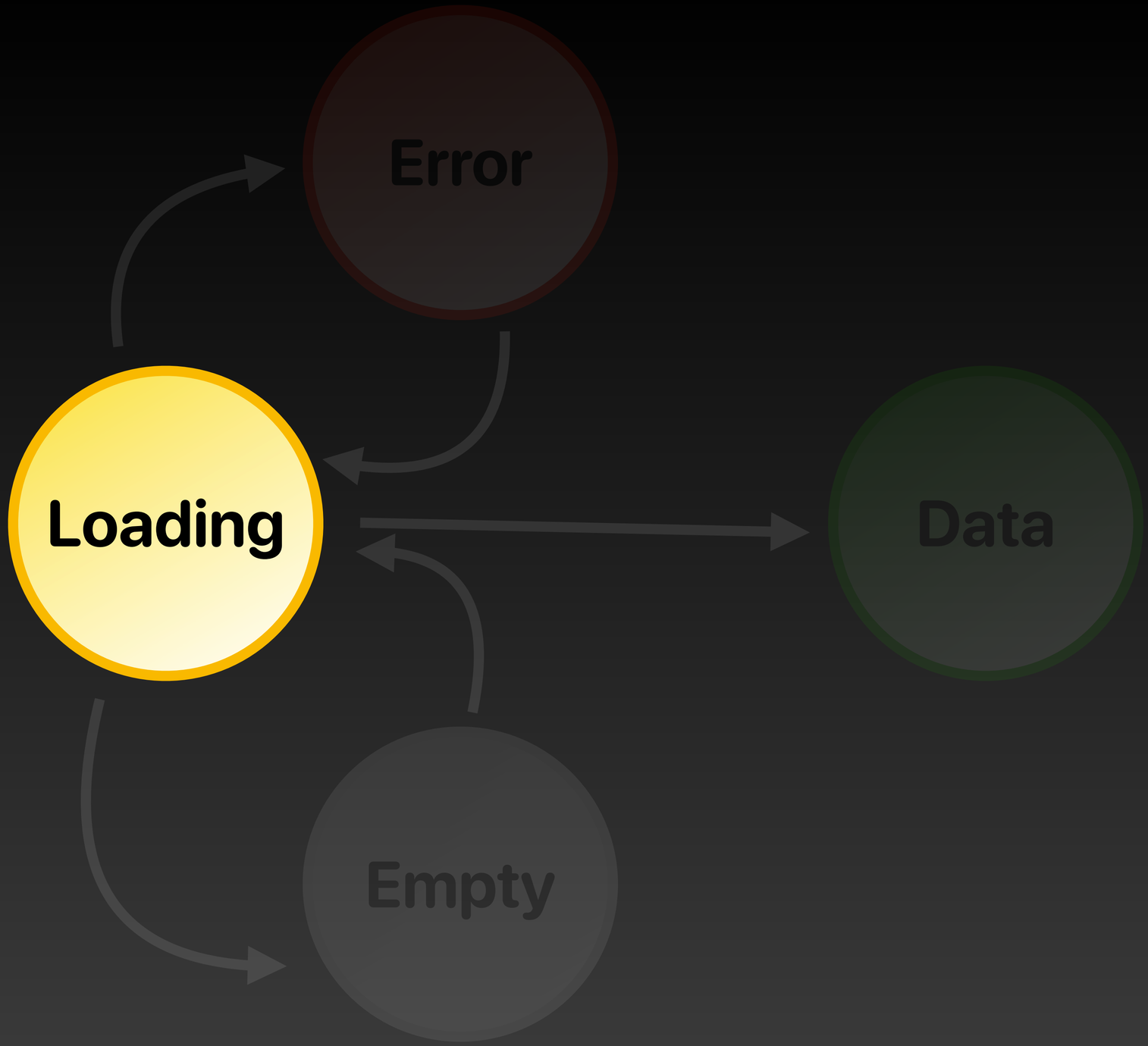
Loading Remote Content

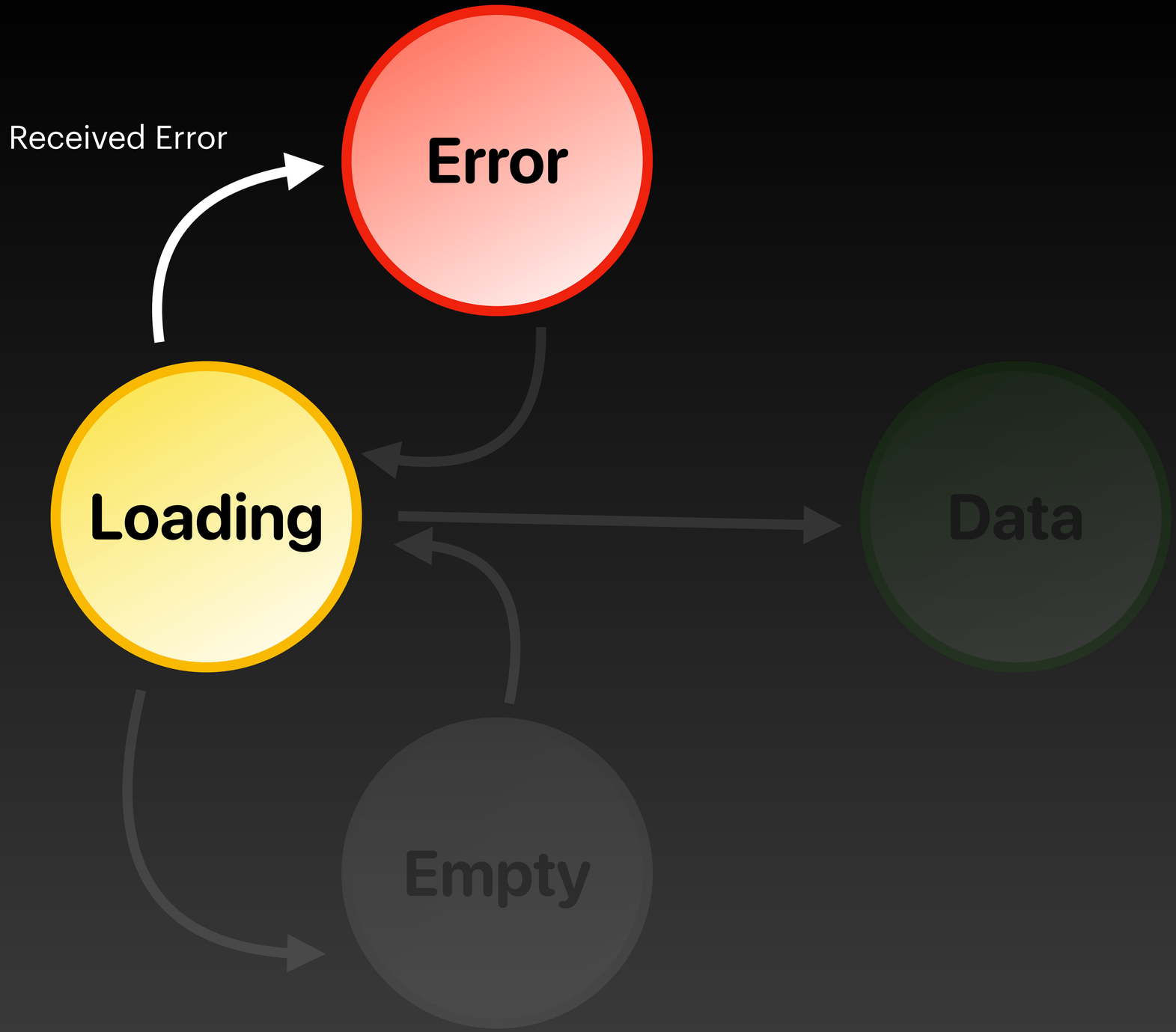

5:34

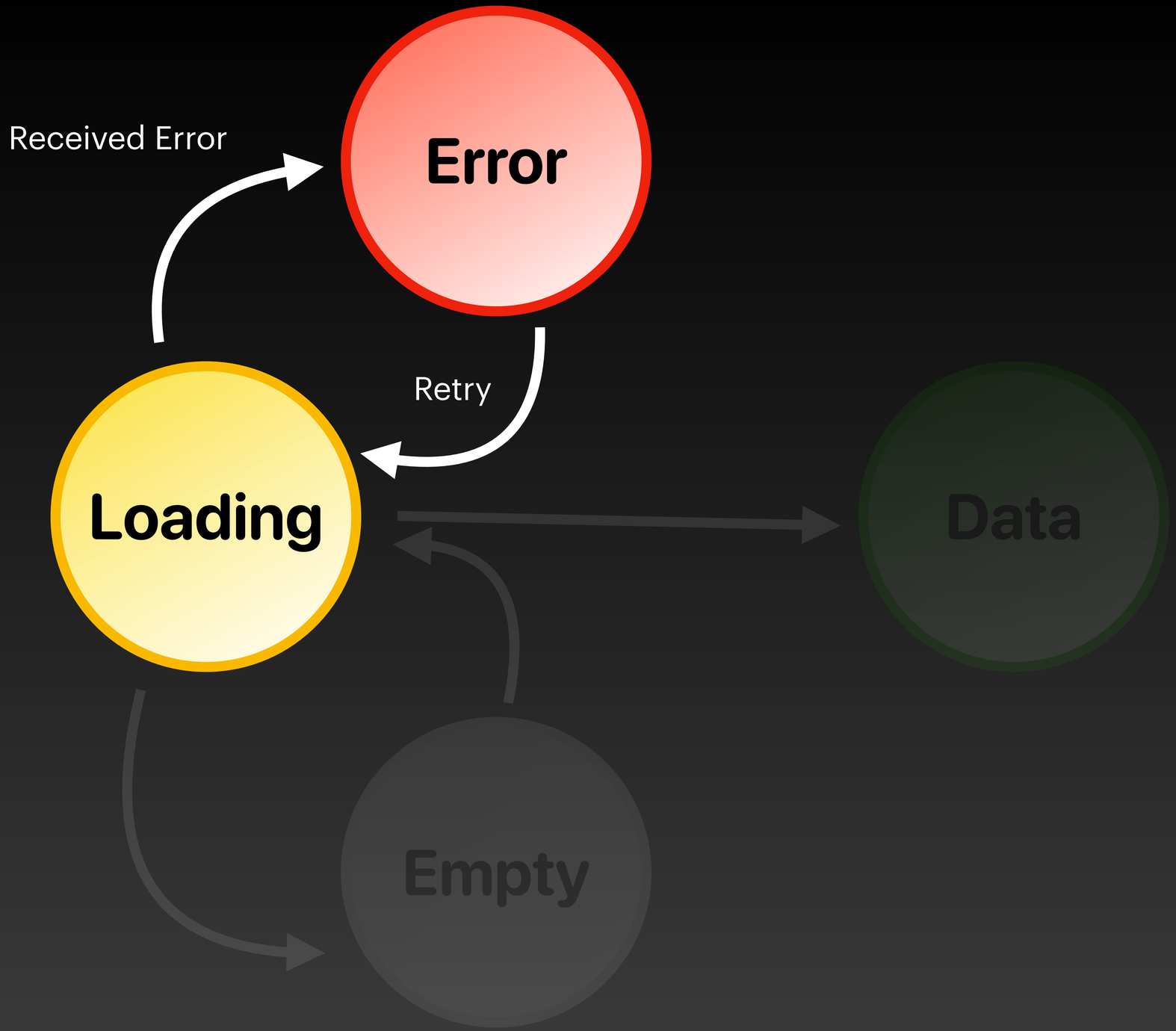

5:33
Error
Retry

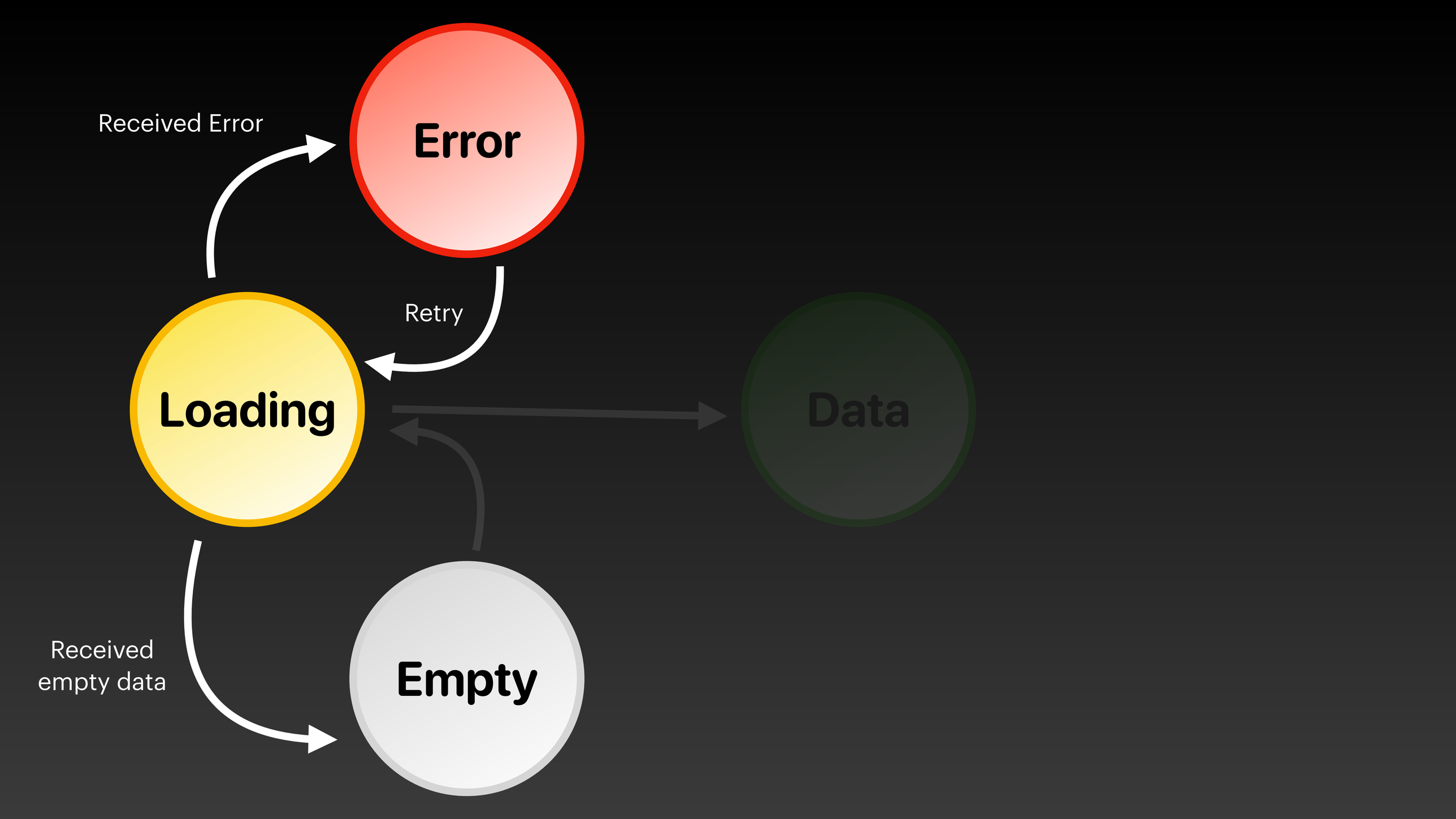

5:45
Success!

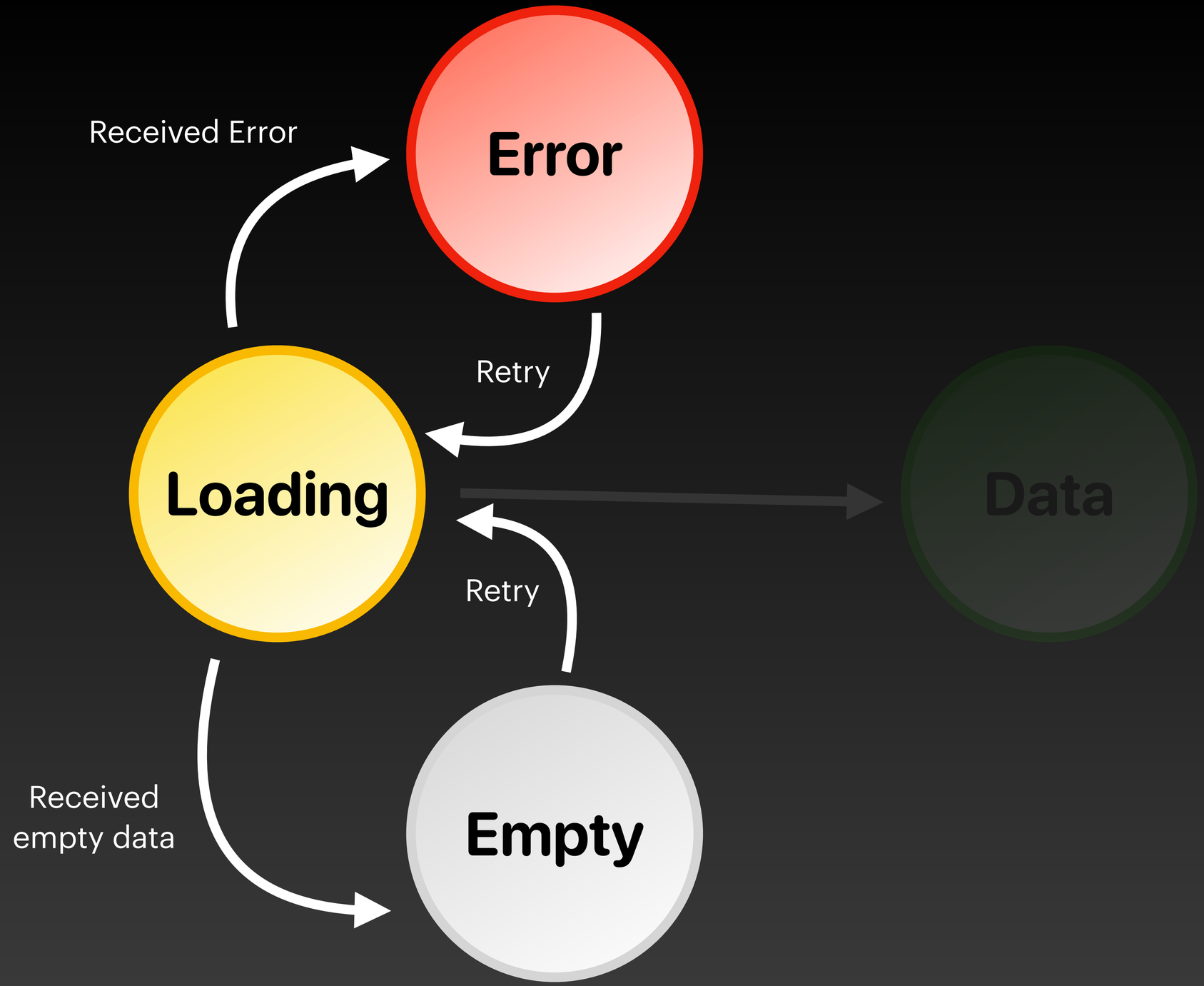

5:33
Empty
Try Again

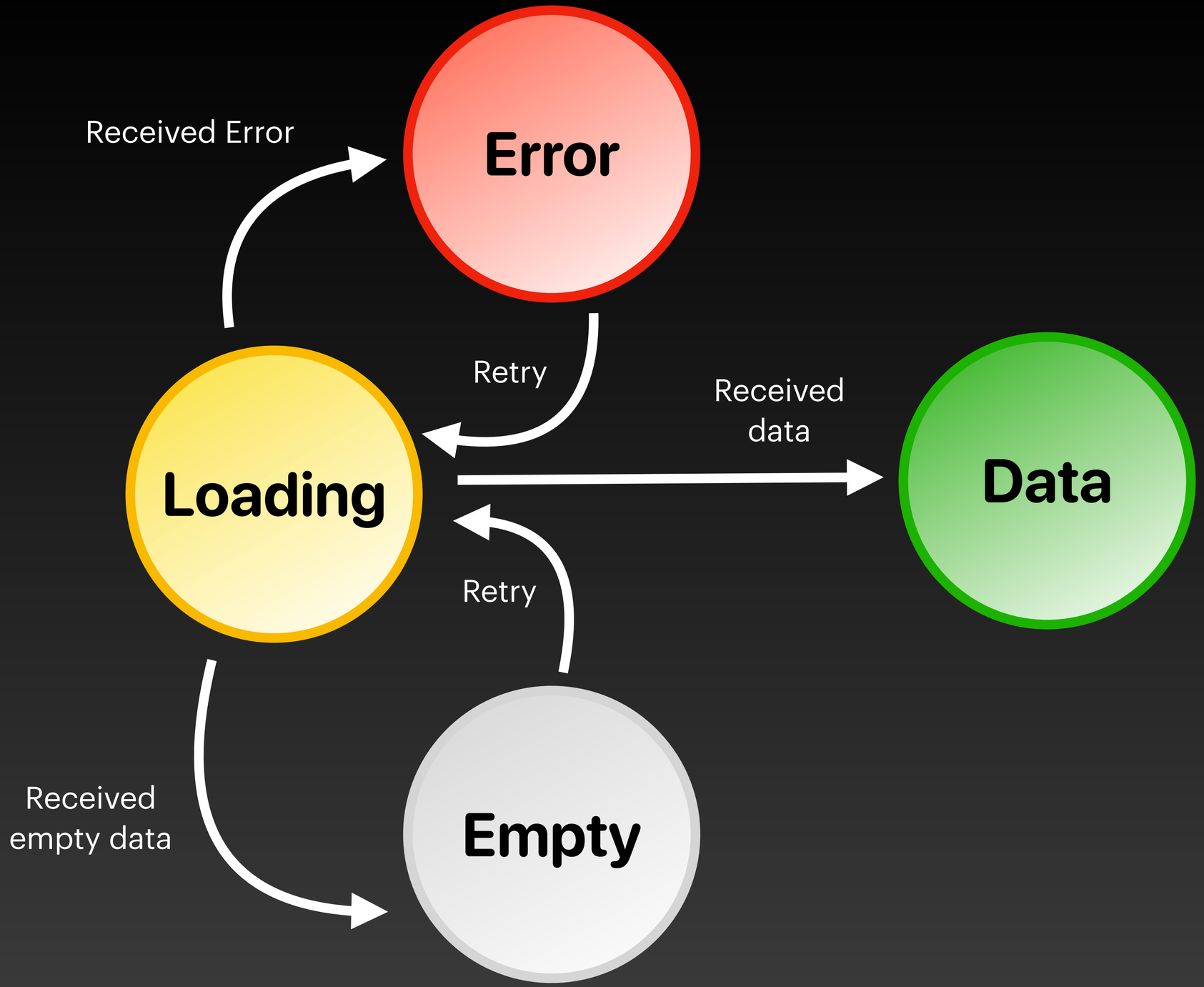

Error
Loading
Data
Empty


Received Error
Error
Loading
Data
Empty


Received Error
Error
Retry
Loading
Data
Empty


Received Error
Error
Retry
Loading
Data
Received
empty data
Empty


Received Error
Error
Retry
Loading
Data
Retry
Received
empty data
Empty


Received Error
Error
Retry
Loading
Received
data
Data
Retry
Received
empty data
Empty

Translating to Swift

The 2 essential parts of a state machine

- A StateDefinition enum
- A StateMachine wrapper

StateDefinition Enum

```
enum RemoteContentStateDefinition {
}
```

StateDefinition Enum

```
enum RemoteContentStateDefinition {
  case loading
  case data
  case error
  case empty
}
```

StateDefinition Enum

```
enum RemoteContentStateDefinition {
  case loading
  case data
  case error
  case empty

  enum Event {
    case didReceiveError
    case didTriggerReload
    case didReceiveEmptyData
    case didReceiveData
  }
}
```

StateDefinition Enum

```
enum RemoteContentStateDefinition {
  //...

  mutating func handleEvent(event: Event) {
  }
}
```

StateDefinition Enum

```
enum RemoteContentStateDefinition {
  //...

  mutating func handleEvent(event: Event) {
    switch (self, event) {
    case (.loading, .didReceiveError):
      self = .error
    case (.loading, .didReceiveData):
      self = .data
    case (.loading, .didReceiveEmptyData):
      self = .empty
    case (.empty, .didTriggerReload):
      self = .loading
    case (.error, .didTriggerReload):
      self = .loading
    }
  }
}
```

StateDefinition Enum

```
enum RemoteContentStateDefinition {
  //...

  mutating func handleEvent(event: Event) {
    switch (self, event) {
    case (.loading, .didReceiveError):
      self = .error
    case (.loading, .didReceiveData):
      self = .data
    case (.loading, .didReceiveEmptyData):
      self = .empty
    case (.empty, .didTriggerReload):
      self = .loading
    case (.error, .didTriggerReload):
      self = .loading
    }
  }
}
```

StateDefinition Enum

StateDefinition Enum

```
enum RemoteContentStateDefinition {
  //...

  mutating func handleEvent(event: Event) {
    switch (self, event) {
    //...
    default:
      print("Invalid state transition from \(self) with event \(event)")
    }
  }
}
```

Part 2: StateMachine class

Part 2: StateMachine class

```
class RemoteContentStateMachine {
  private var state = RemoteContentStateDefinition.loading
}
```

Part 2: StateMachine class

```
protocol RemoteContentStateMachineDelegate: AnyObject {
  func didChangeState(
    _ state: RemoteContentStateDefinition,
    in stateMachine: RemoteContentStateMachine
  )
}

class RemoteContentStateMachine {
  weak var delegate: RemoteContentStateMachineDelegate?

  private var state = RemoteContentStateDefinition.loading {
    didSet {
      delegate?.didChangeState(state, in: self)
    }
  }
}
```

Part 2: StateMachine class

```
class RemoteContentStateMachine {
  //...

  func receiveError() { }

  func reload() { }

  func receiveData(data: [Any]) { }
}
```

Part 2: StateMachine class

```
class RemoteContentStateMachine {
  //...

  func receiveError() {
    state.handleEvent(event: .didReceiveError)
  }

  func reload() {
    state.handleEvent(event: .didTriggerReload)
  }

  func receiveData(data: [Any]) {
    if data.isEmpty {
      state.handleEvent(event: .didReceiveEmptyData)
    } else {
      state.handleEvent(event: .didReceiveData)
    }
  }
}
```

Part 2: StateMachine class

```
class RemoteContentStateMachine {
  func start() {
    delegate?.didChangeState(state, in: self)
  }
}
```

Integrating into our app

```
extension RemoteContentContainerController: RemoteContentStateMachineDelegate {
  func didChangeState(
    _ state: RemoteContentStateDefinition,
    in stateMachine: RemoteContentStateMachine
  ) {
    switch state {
    case .loading:
      switchTo(LoadingViewController())
      fetch()
    case .data:
      switchTo(DataViewController())
    case .error:
      let errorViewController = ErrorViewController()
      errorViewController.delegate = self
      switchTo(errorViewController)
    case .empty:
      let emptyViewController = EmptyViewController()
      emptyViewController.delegate = self
      switchTo(emptyViewController)
    }
  }
}
```

Child State Machines

Child State Machines

- In some use cases, child state machines can grow over time
- Coding large state machines is no fun

Child State Machine

```
enum DataPresenceStateDefinition {
  case empty
  case data
  case refresh
}
```

Child State Machine

```
enum DataPresenceStateDefinition {
  enum Event {
    case didTriggerRefresh
    case didReceiveEmptyData
    case didReceiveData
  }
}
```

Child State Machine

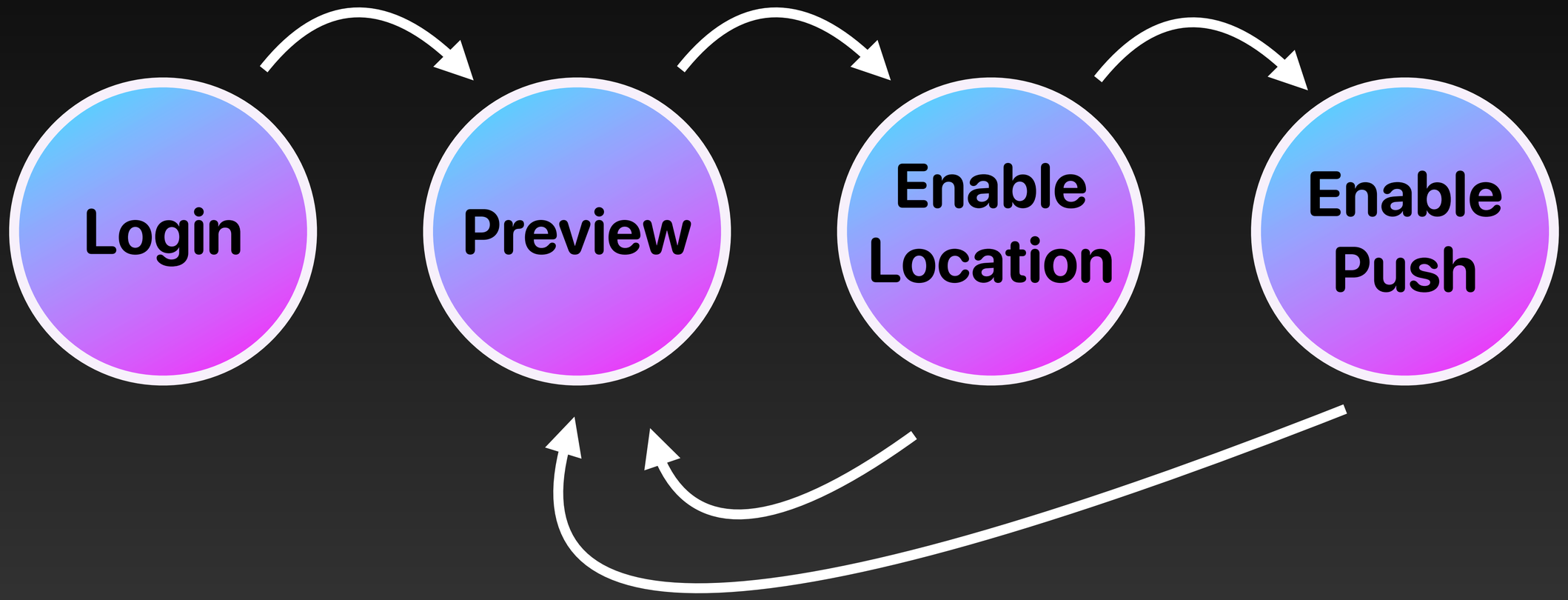
```
enum DataPresenceStateDefinition {
  mutating func handleEvent(event: Event) {
    switch (self, event) {
    case (.data, .didTriggerRefresh):
      self = .refresh
    case (.empty, .didTriggerRefresh):
      self = .refresh
    case (.refresh, .didReceiveData):
      self = .data
    case (.refresh, .didReceiveEmptyData):
      self = .empty
    case (.refresh, .didTriggerRefresh):
      self = .refresh
    default:
      print("Invalid state transition")
    }
  }
}
```

StateDefinition Enum

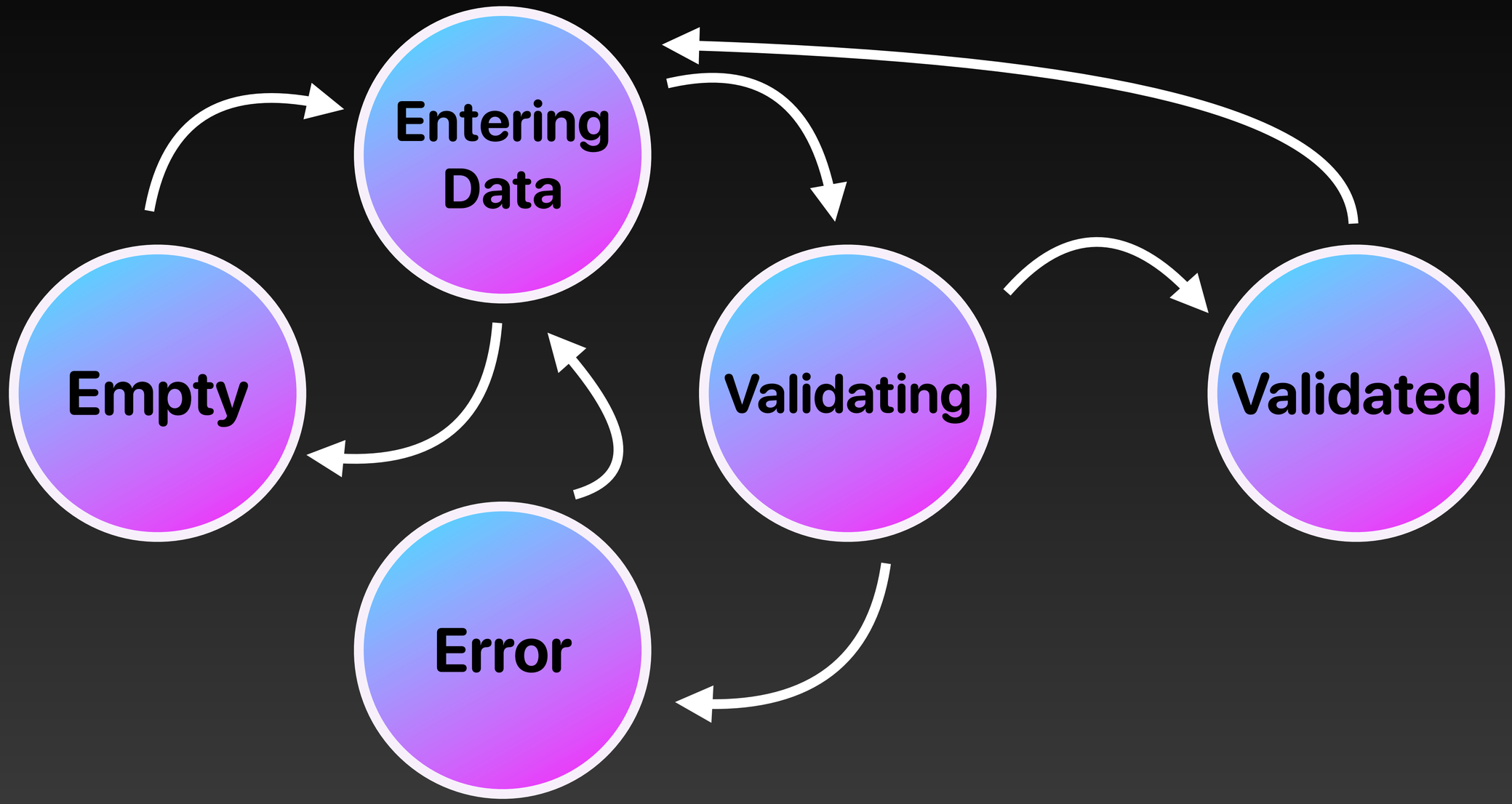
```
enum RemoteContentStateDefinition {
  case loading
  case loaded(DataPresenceStateDefinition)
  case error
}
```

Examples

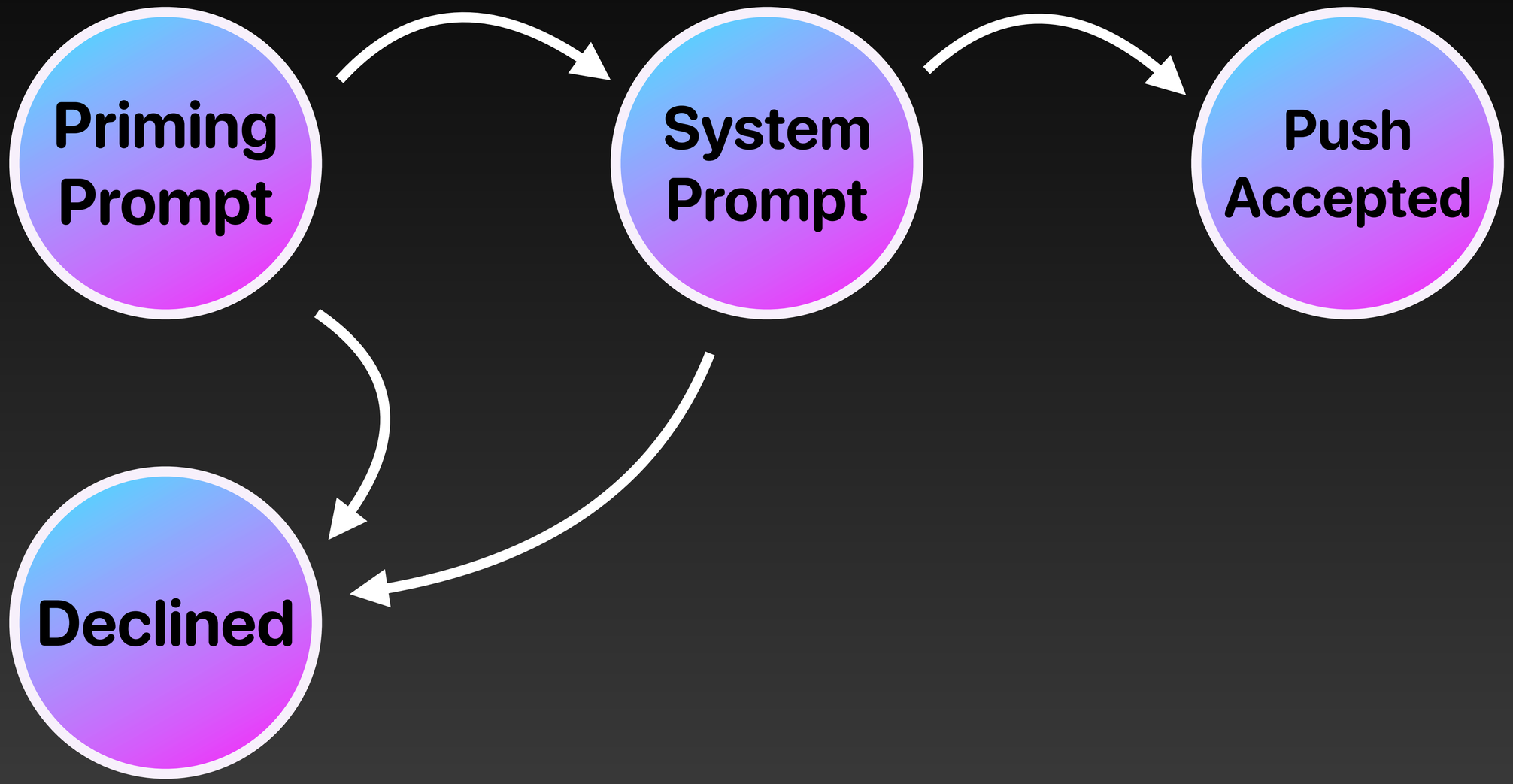
Onboarding Coordinators



Form Validation



Accepting Push Notifications



Sample Code

Get the sample code at:

bit.ly/state-machines-conf42

Other Resources

- This gist by Andy Matuschak
- Building State Machines in Swift by Cory Benfield
- State Machines are your friend by Matt Delves

Contact!

Don't be shy, say hi!

- @frankacy on Twitter
- @frankacy in Slack
- hello@frankcourville.com