App Frameworks

#WWDC16

# CloudKit Best Practices

Session 231

Dave Browning CloudKit Engineer
Nihar Sharma CloudKit Engineer

## What We'll Cover

## What We’ll Cover

How Apple uses CloudKit

## What We’ll Cover

How Apple uses CloudKit

Using the CKOperation API

## What We'll Cover

How Apple uses CloudKit

Using the CKOperation API

Modeling your data

## What We’ll Cover

How Apple uses CloudKit

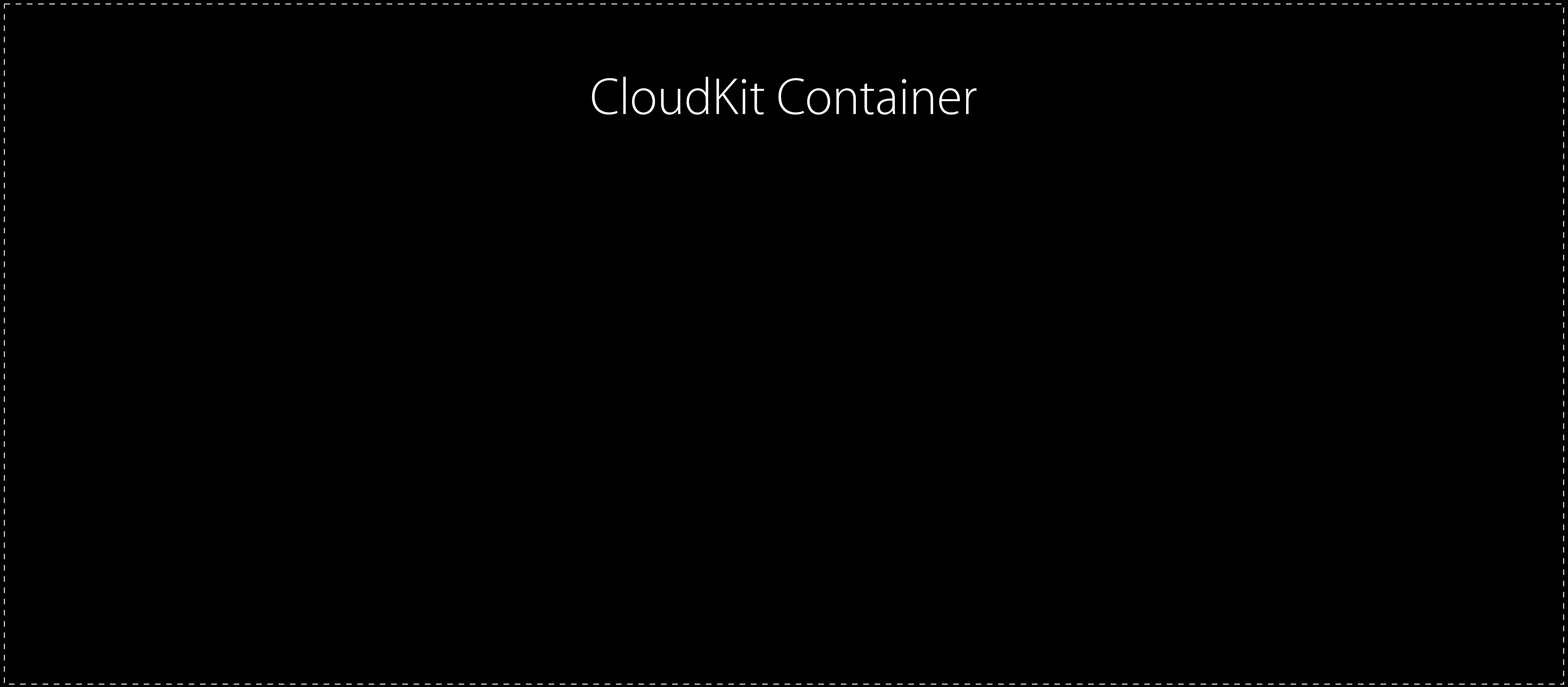
Using the CKOperation API

Modeling your data

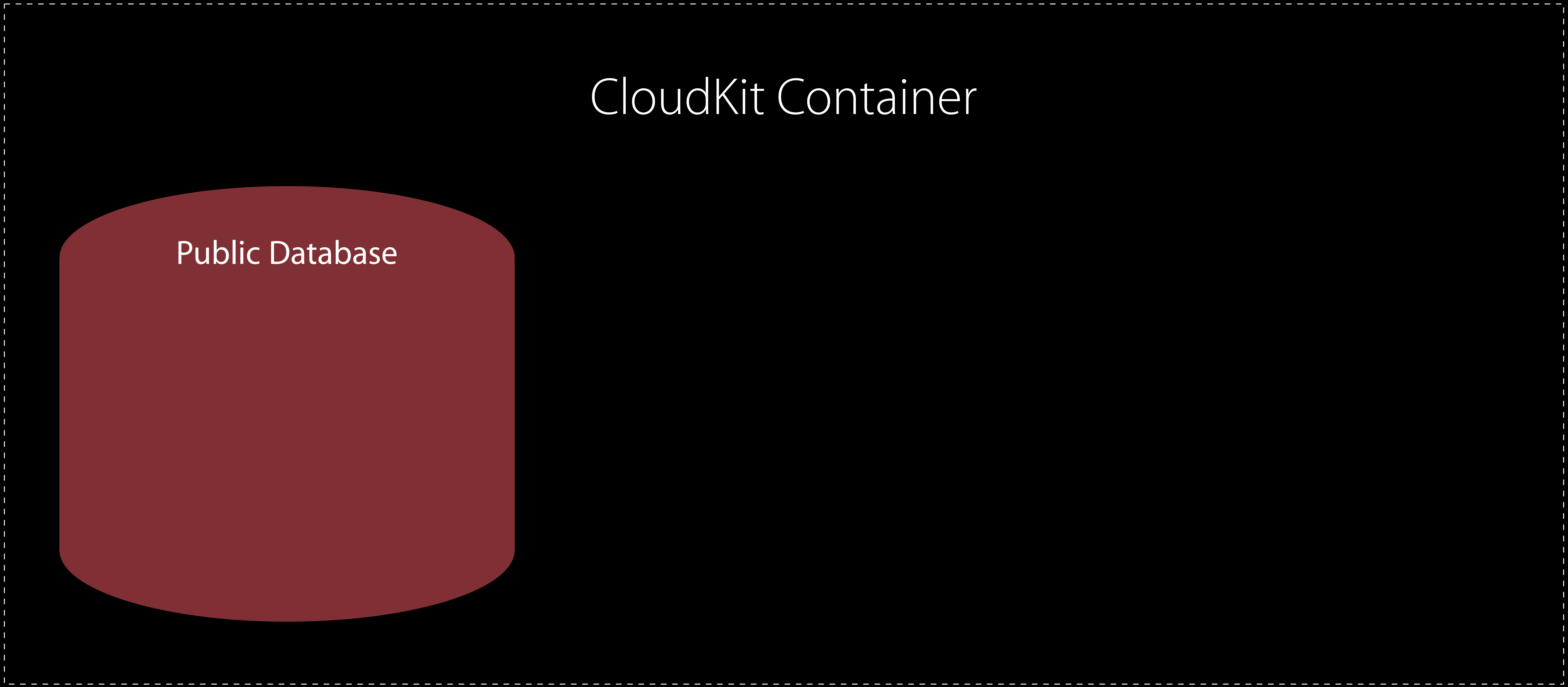
Handling errors

## Quick Refresher

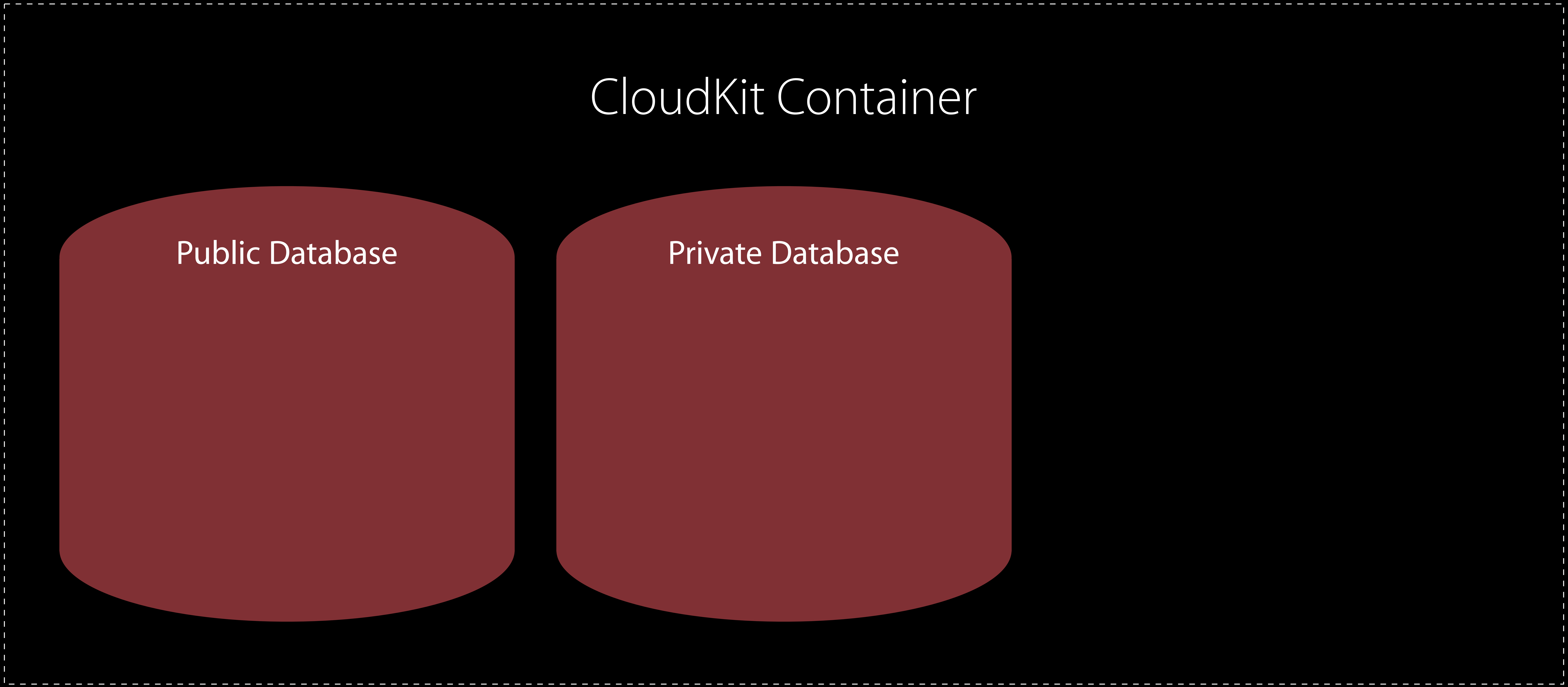
## Quick Refresher



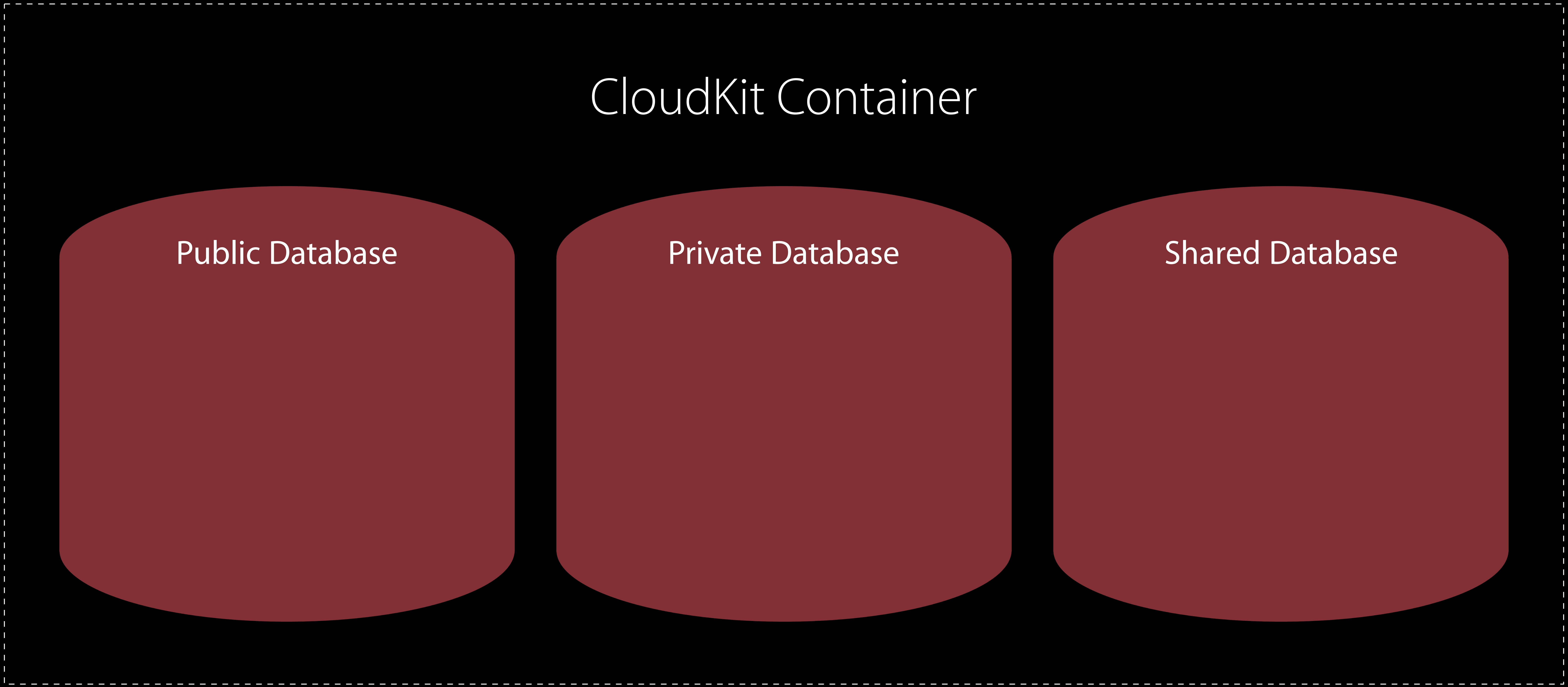
## Quick Refresher



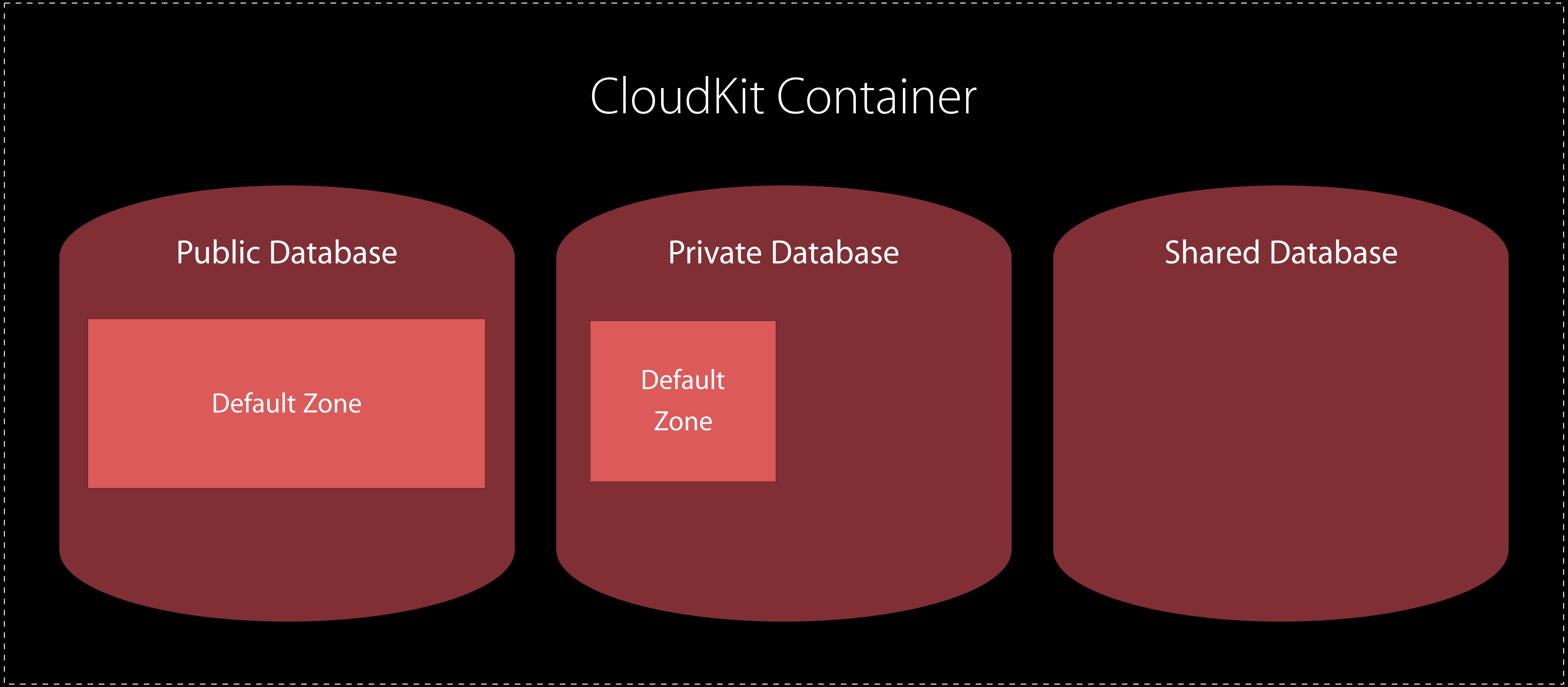
## Quick Refresher



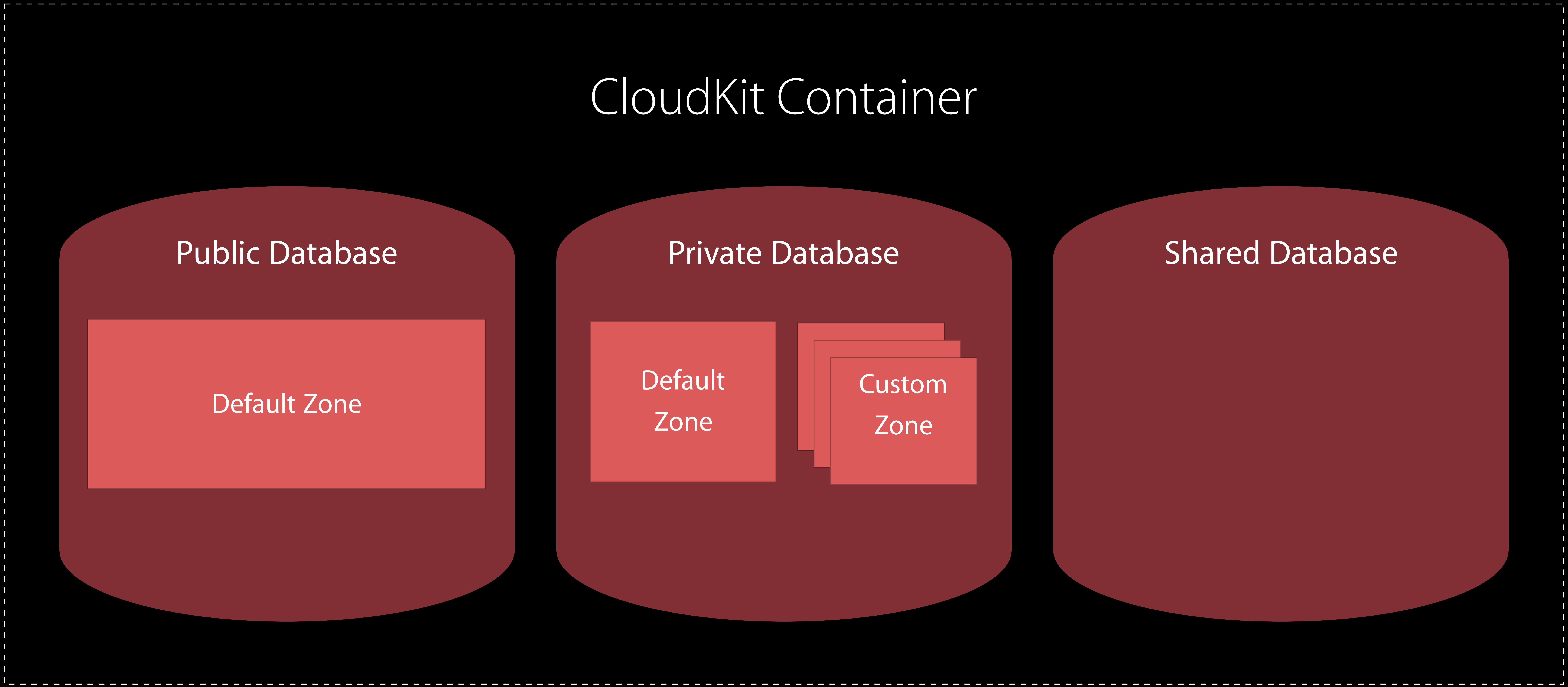
## Quick Refresher



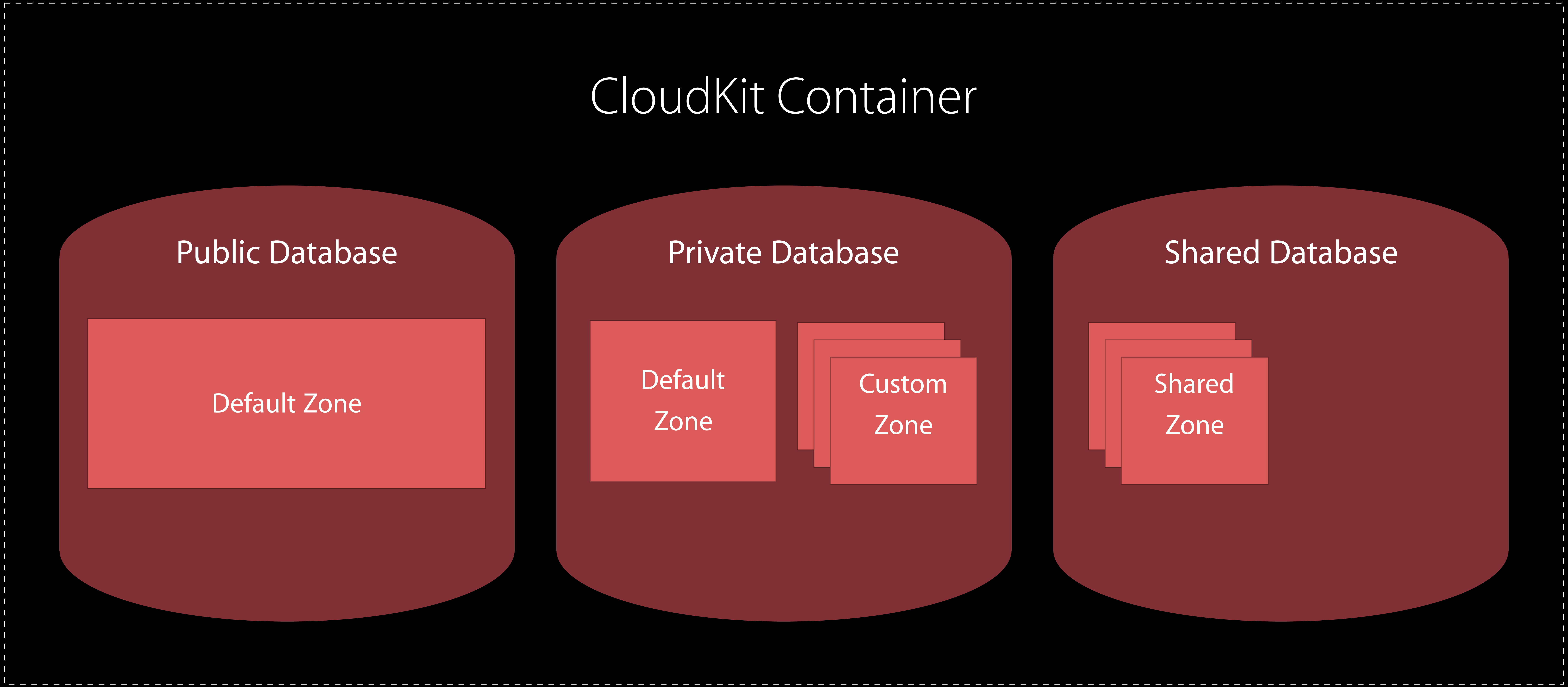
## Quick Refresher



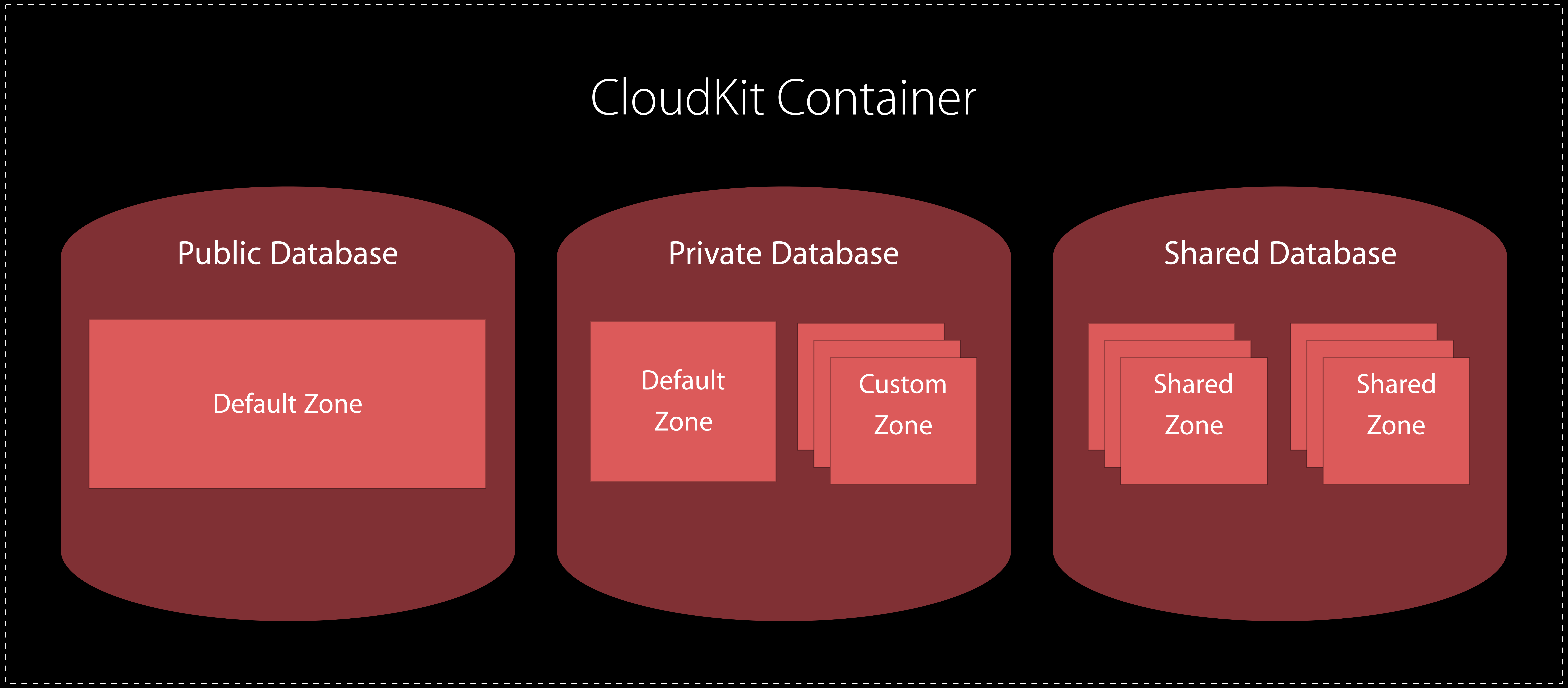
## Quick Refresher



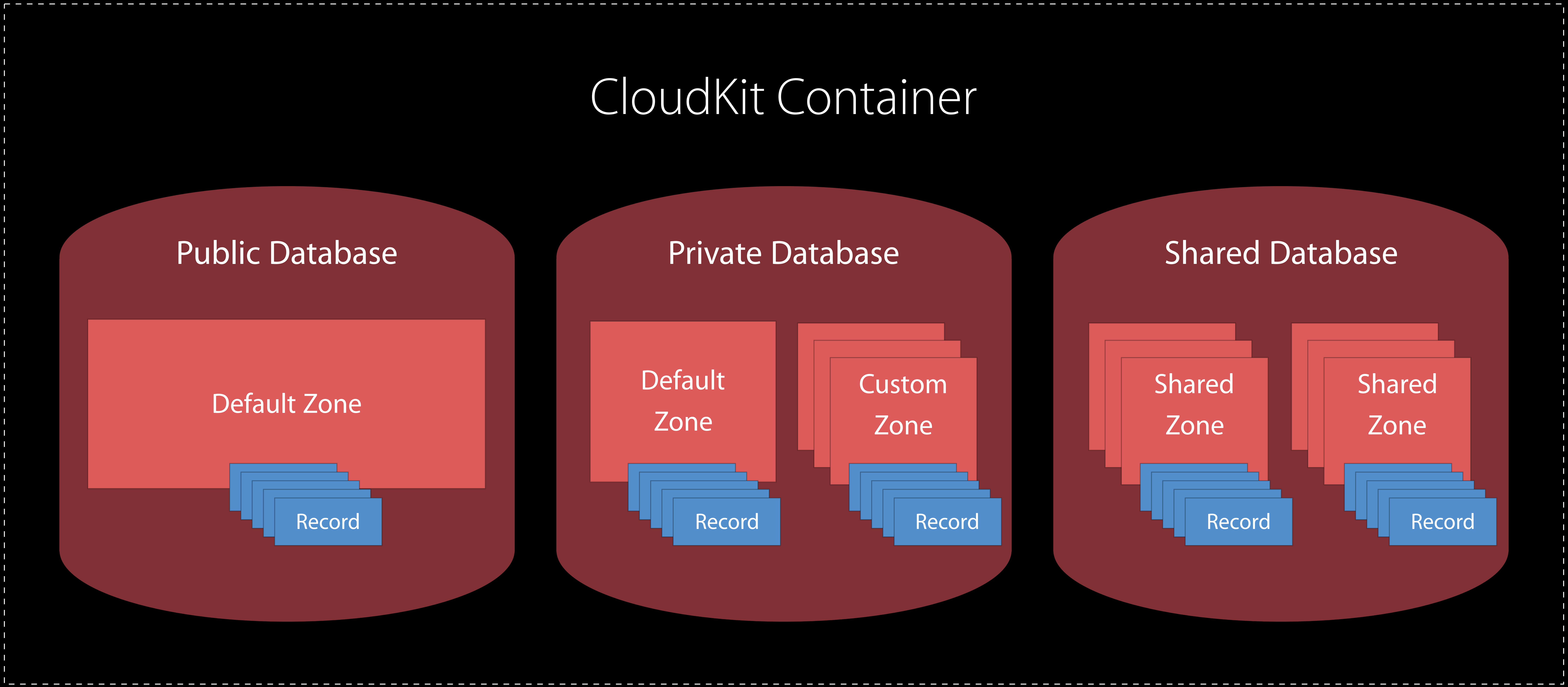
# Quick Refresher



## Quick Refresher



# Quick Refresher



✓ Focus on building your app

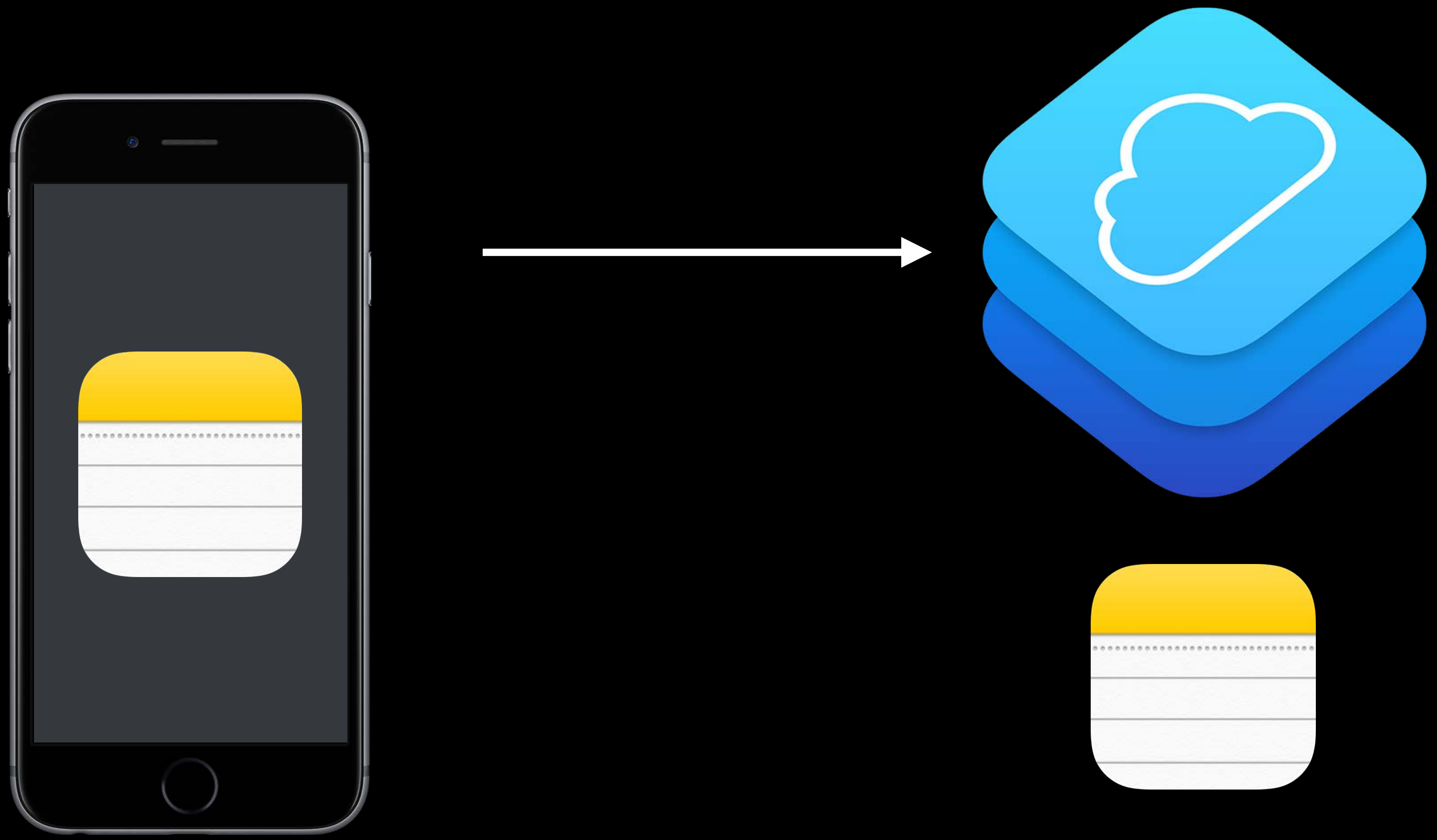
- ✓ Focus on building your app
- ✓ Automatic authentication

- ✓ Focus on building your app
- ✓ Automatic authentication
- ✓ Same data across all devices

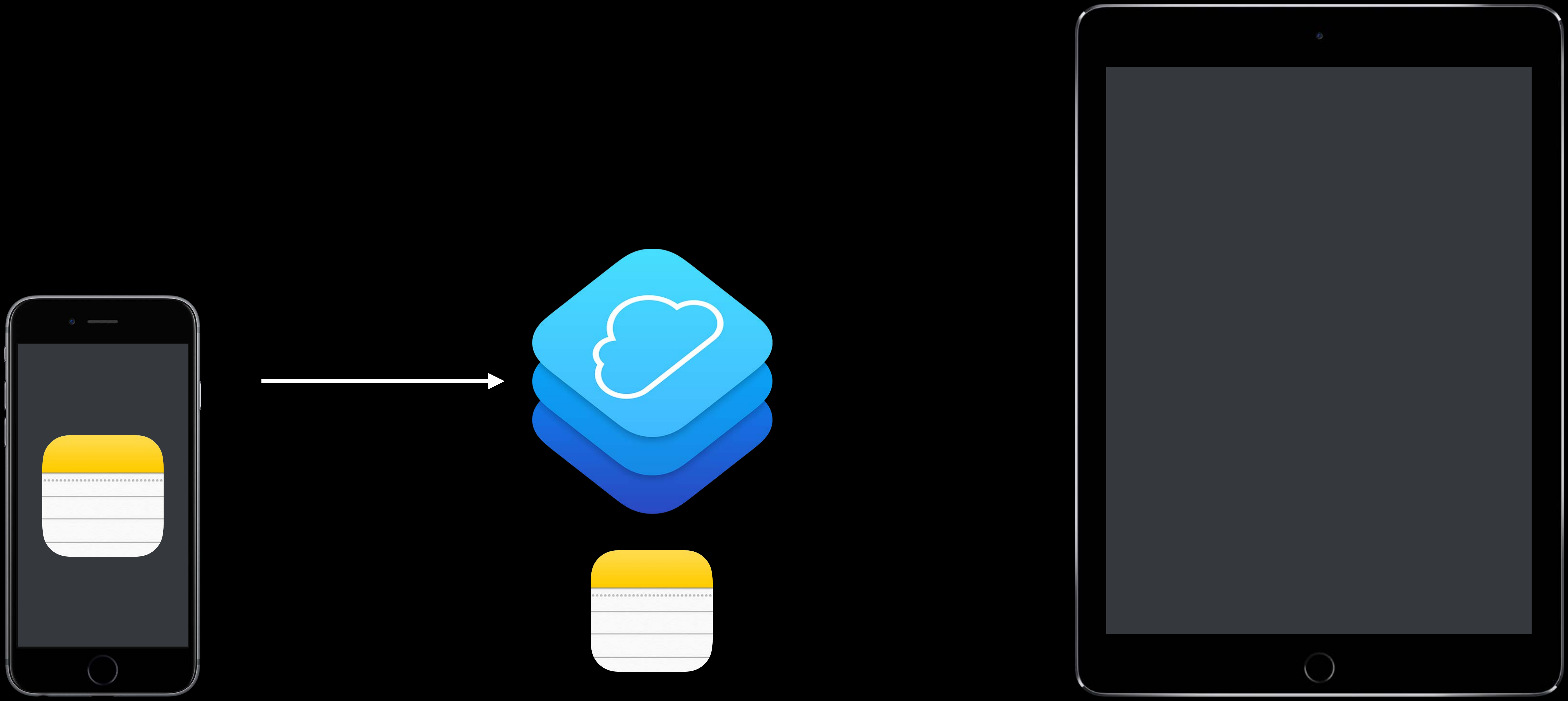
# The Use Case

## The Use Case

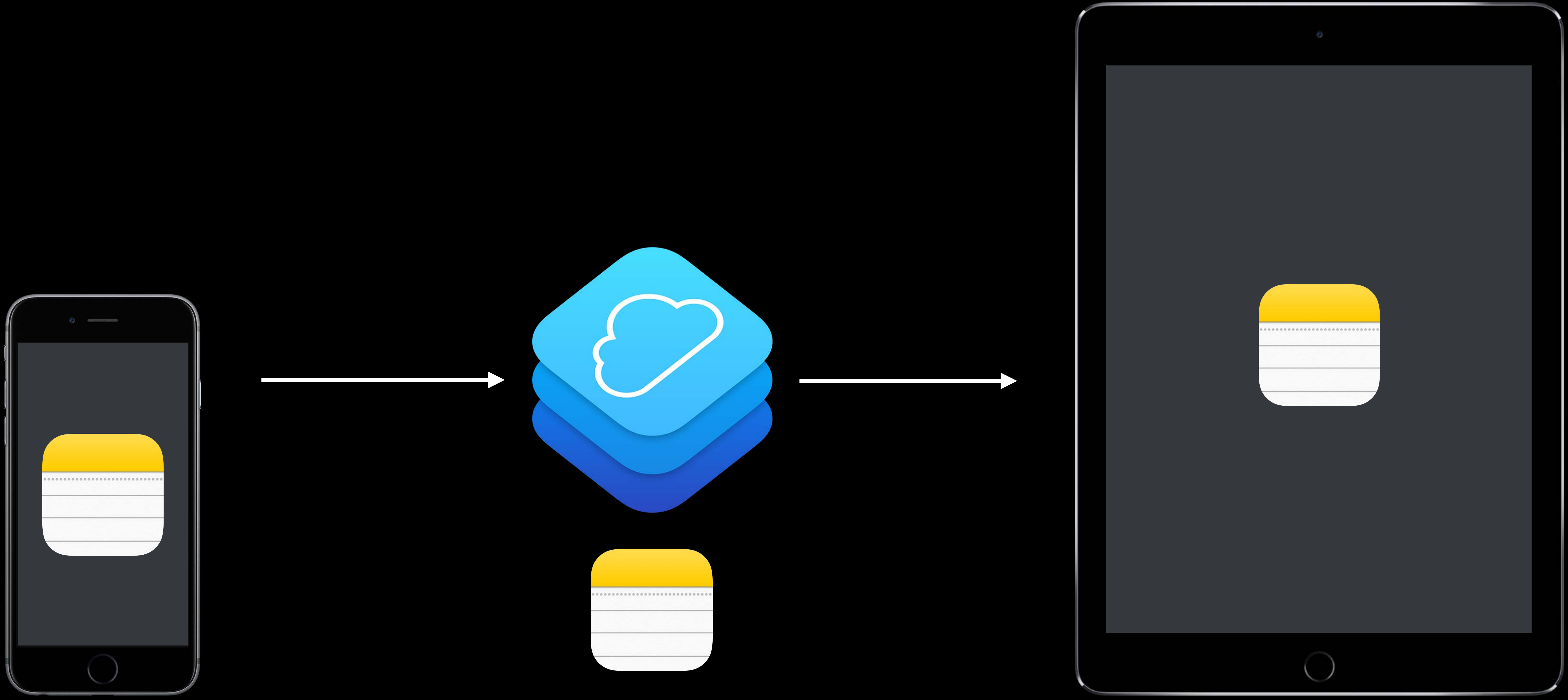
## The Use Case



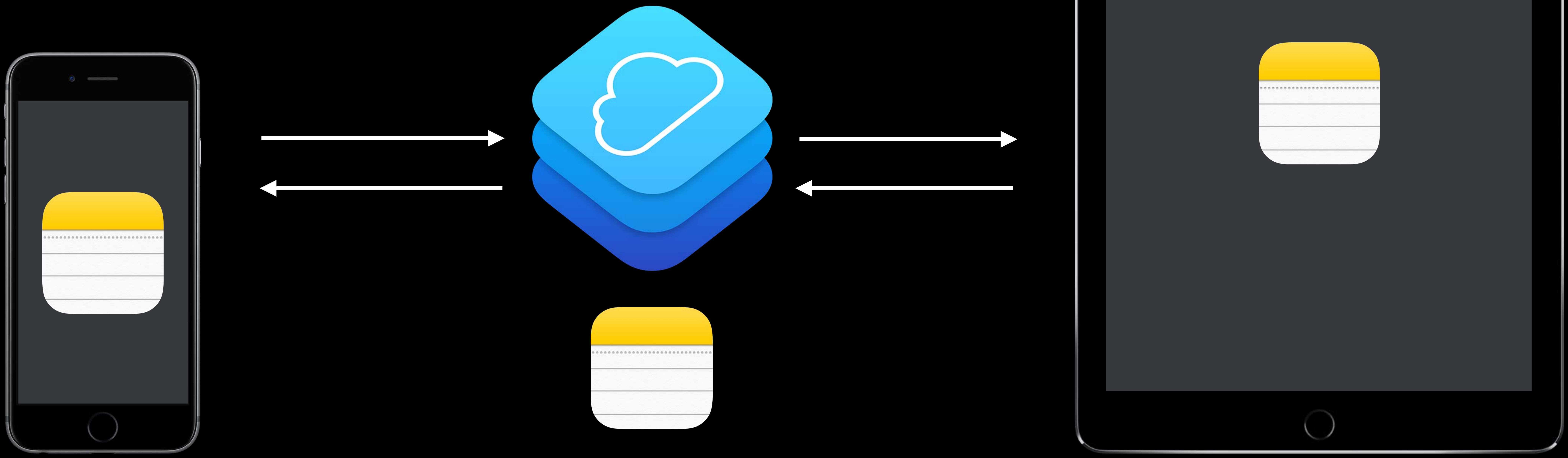
## The Use Case



## The Use Case



## The Use Case



✓ iCloud is the source of truth

- ✓ iCloud is the source of truth
- ✓ Devices have a local cache

- ✓ iCloud is the source of truth
- ✓ Devices have a local cache
- ✓ CloudKit is the glue

## How Does It Work?

## On app launch

On app launch

Fetch changes

## On app launch

- Fetch changes
- Subscribe to future changes

On app launch

- Fetch changes
- Subscribe to future changes

On push from CloudKit

On app launch

- Fetch changes
- Subscribe to future changes

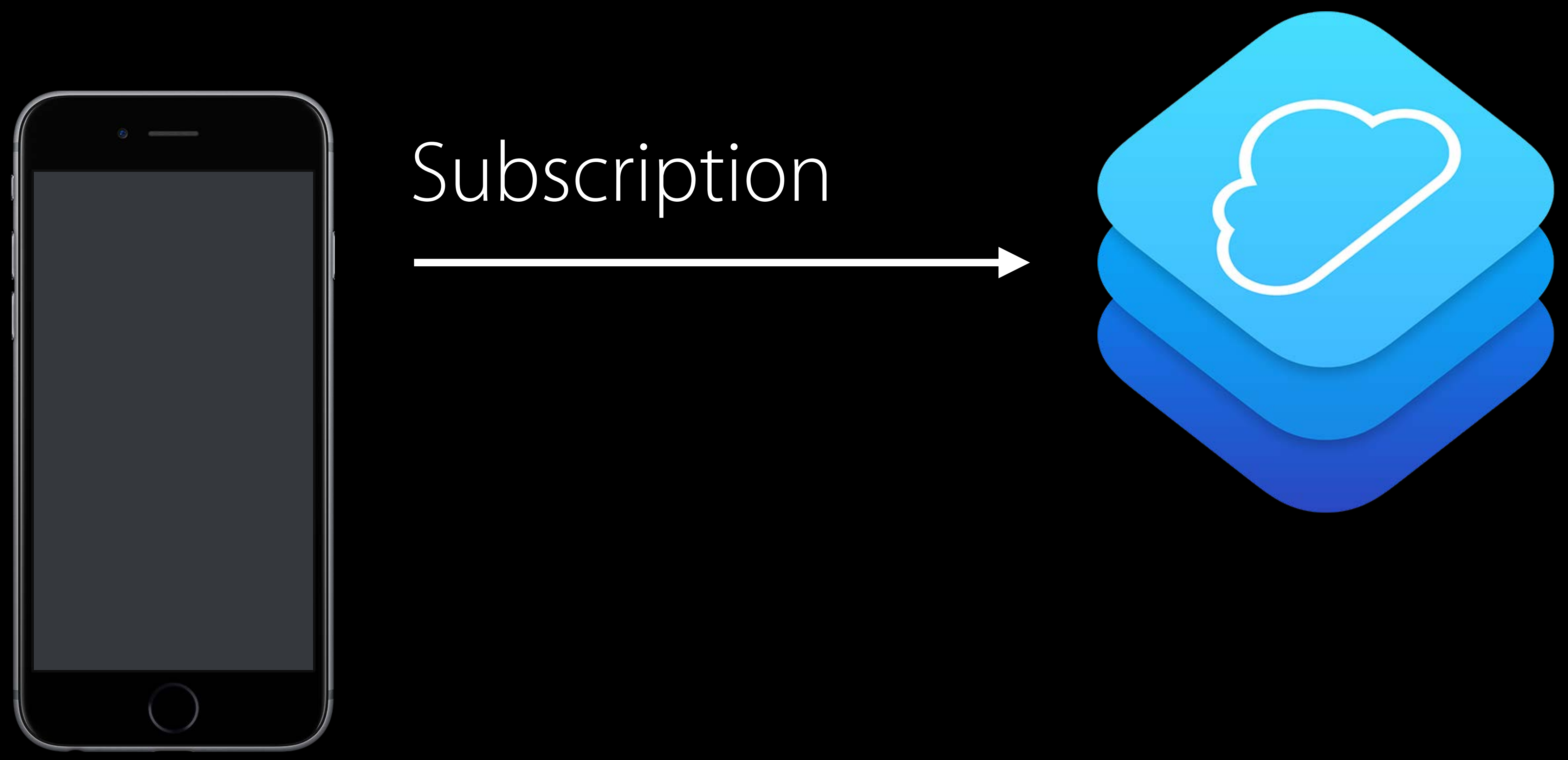
On push from CloudKit

- Fetch changes

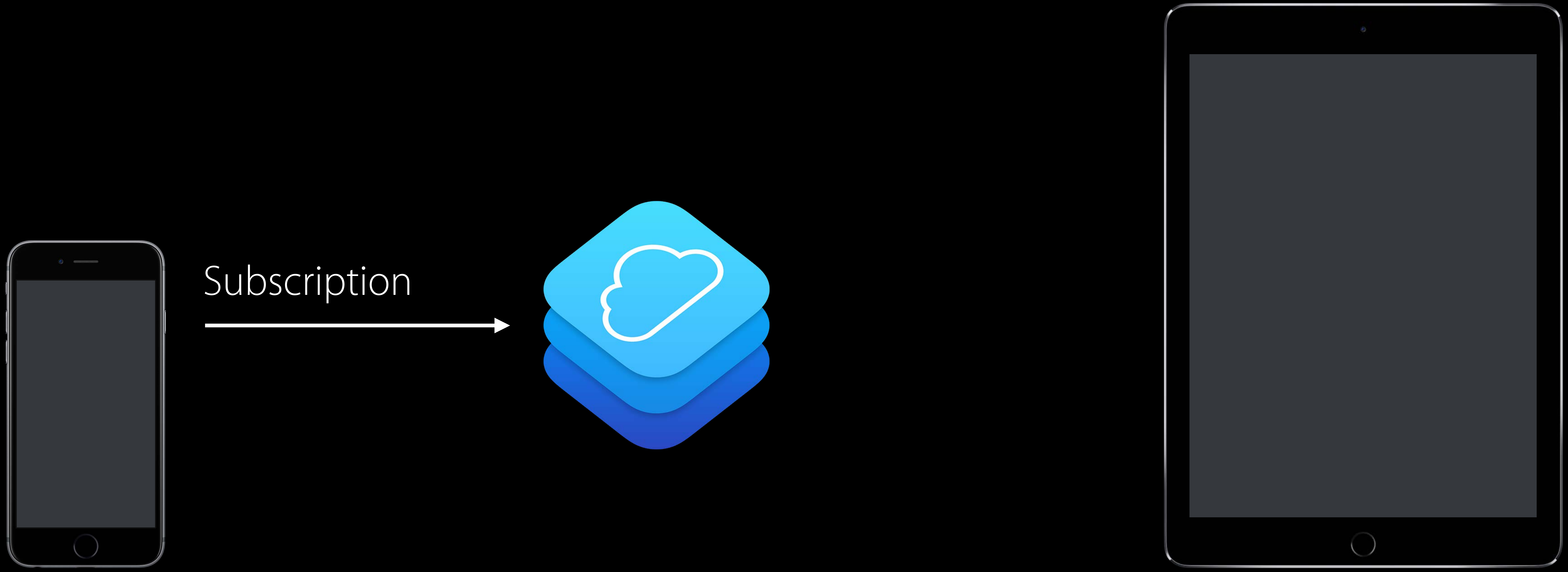
## Subscribe to Changes

# Subscribe to Changes

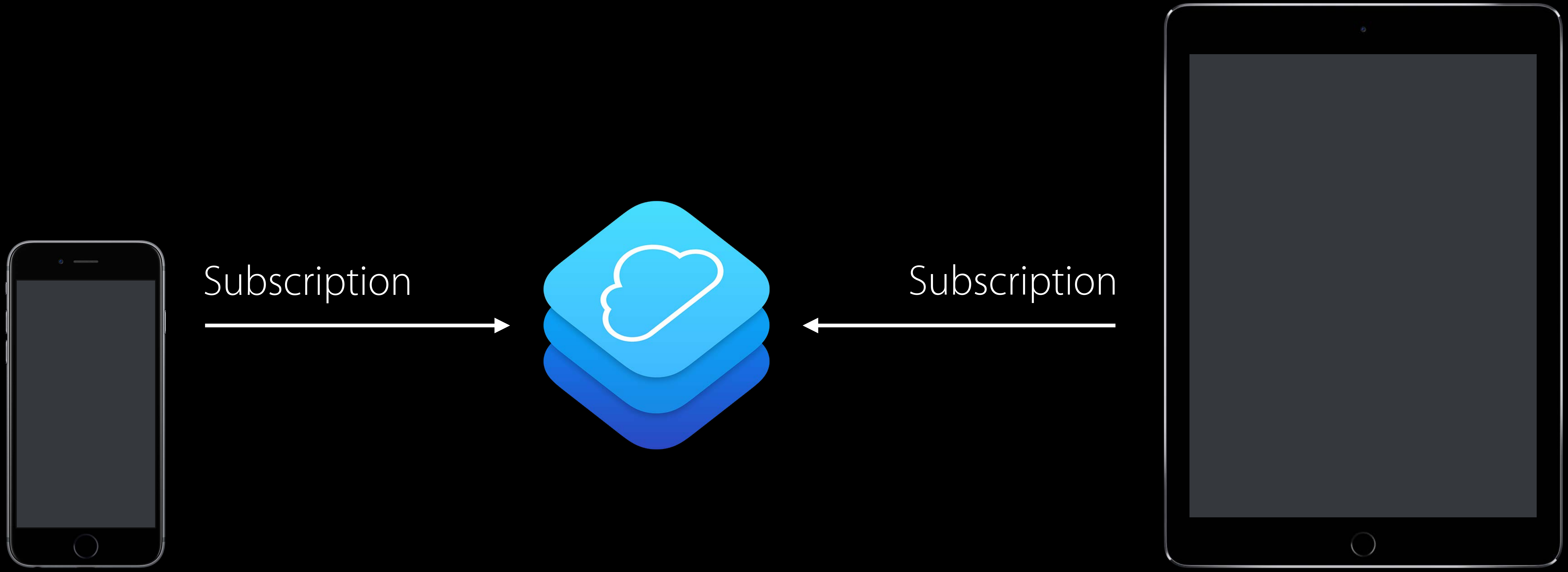
# Subscribe to Changes



# Subscribe to Changes



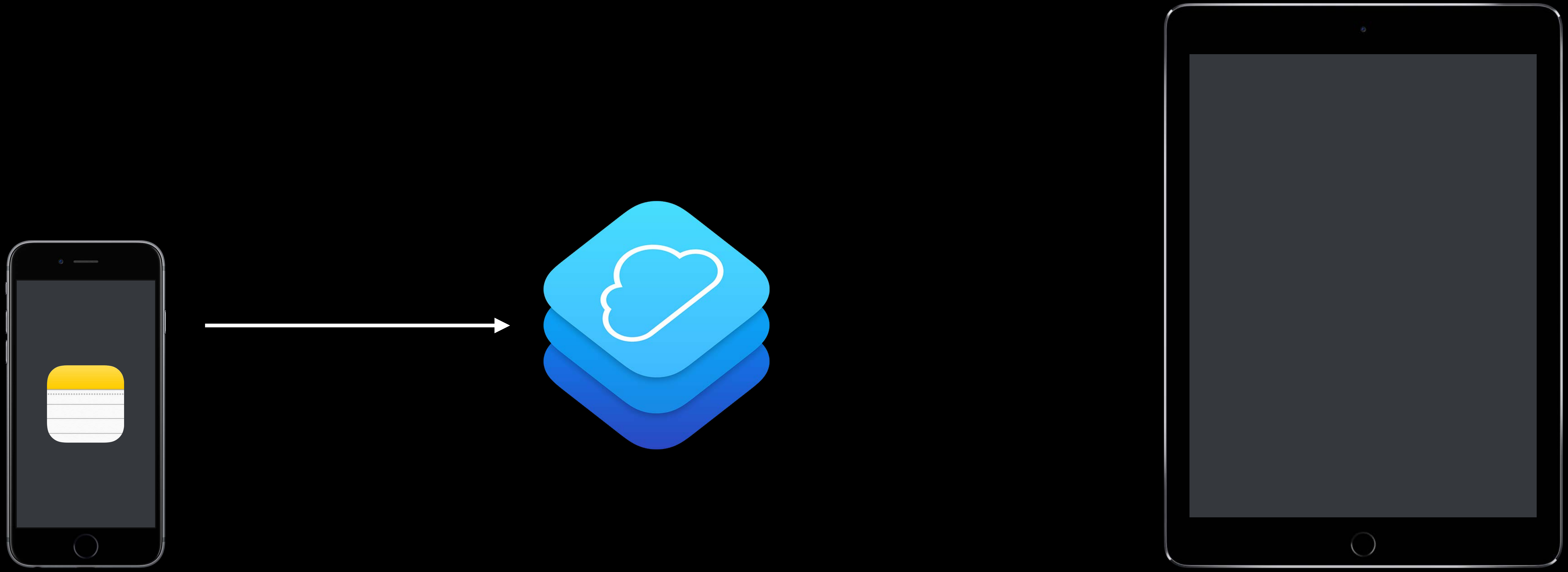
# Subscribe to Changes



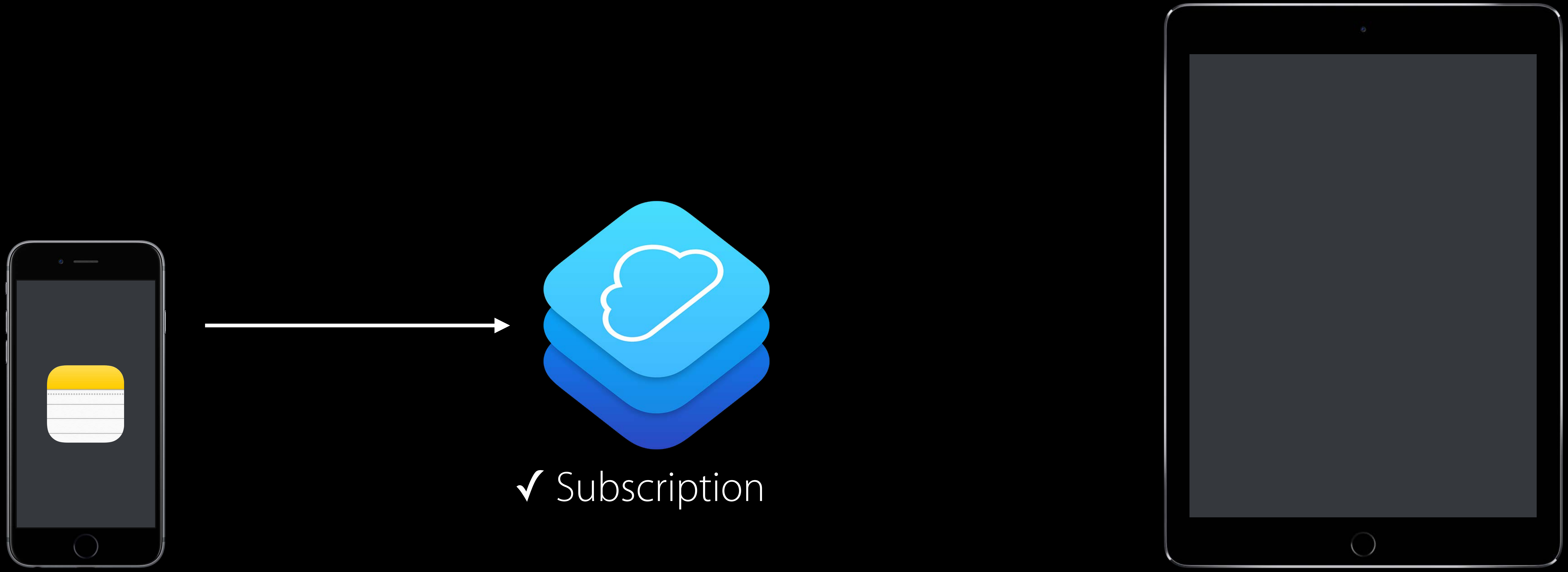
## Listen for Pushes

## Listen for Pushes

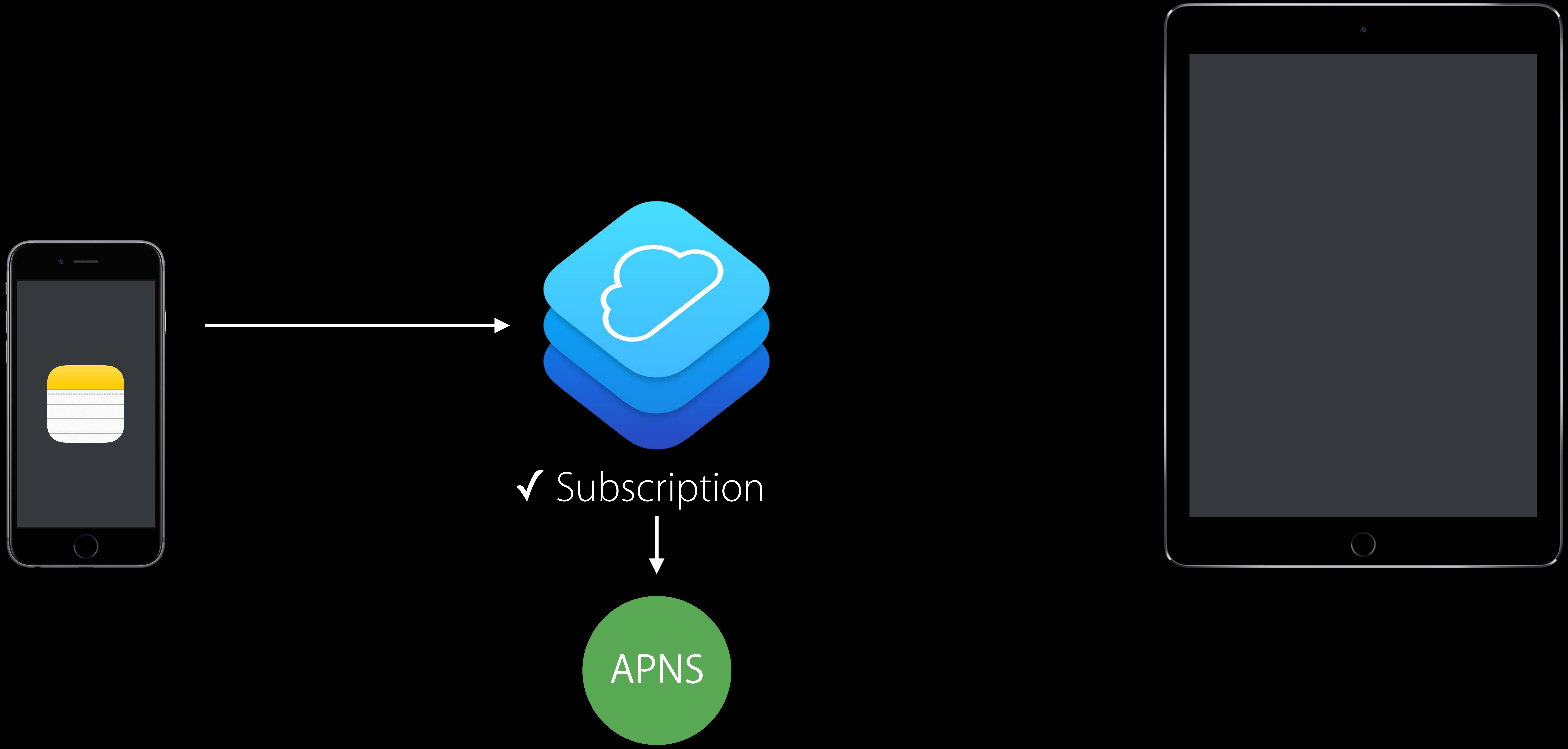
## Listen for Pushes



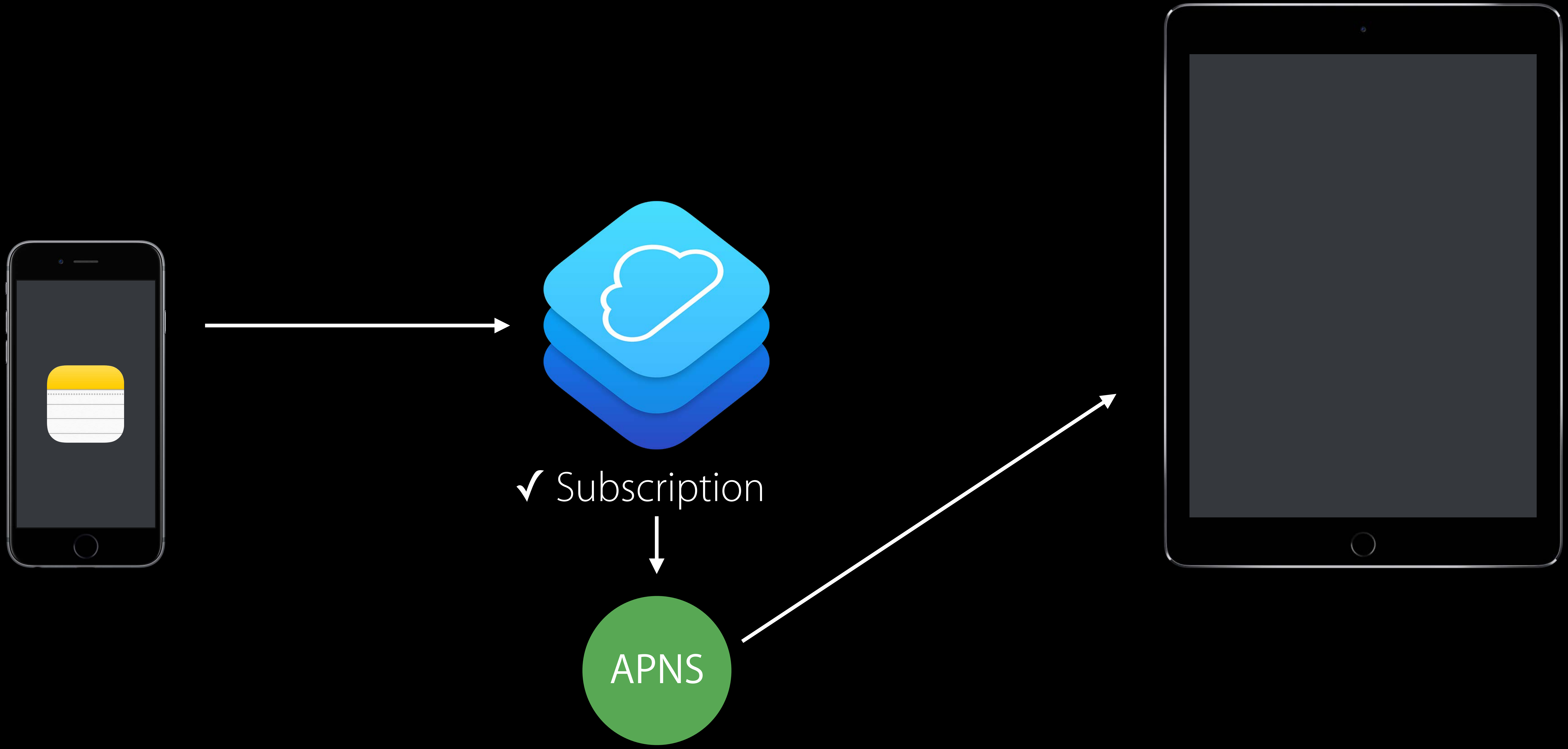
## Listen for Pushes



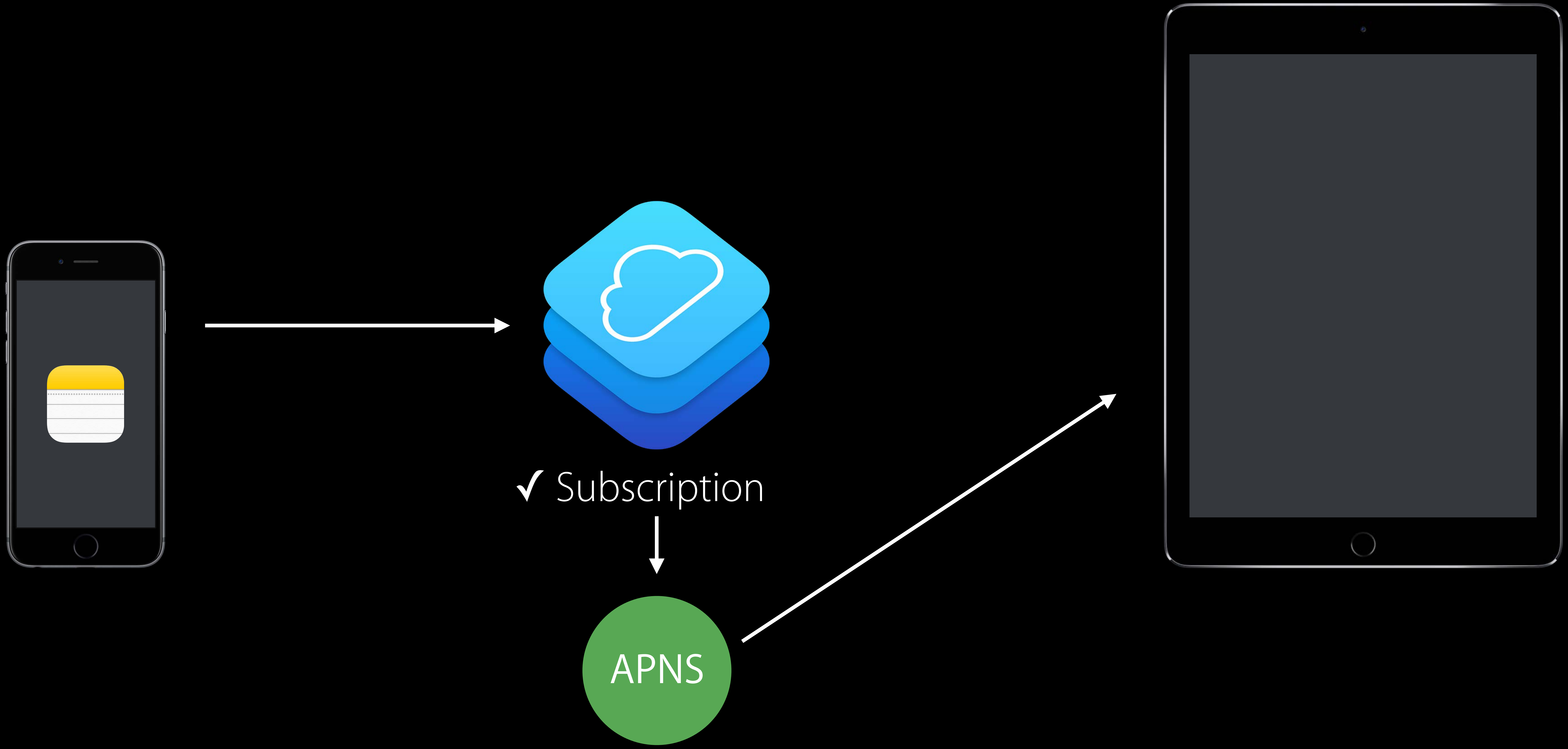
## Listen for Pushes



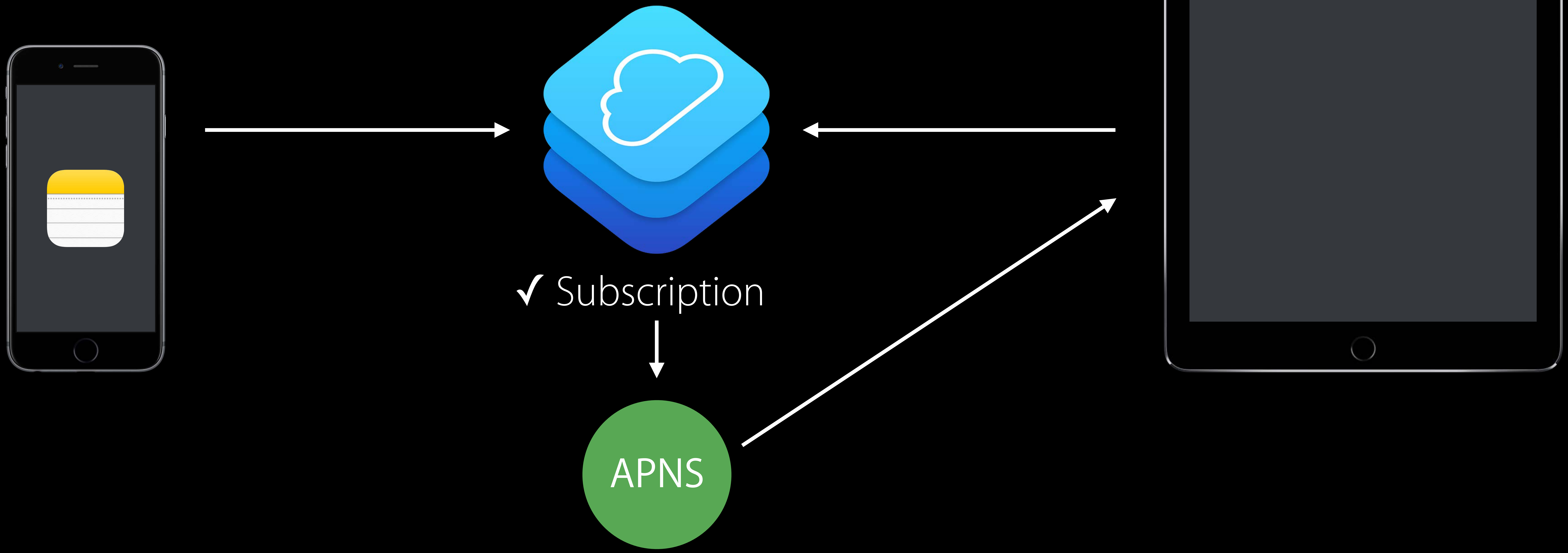
# Listen for Pushes



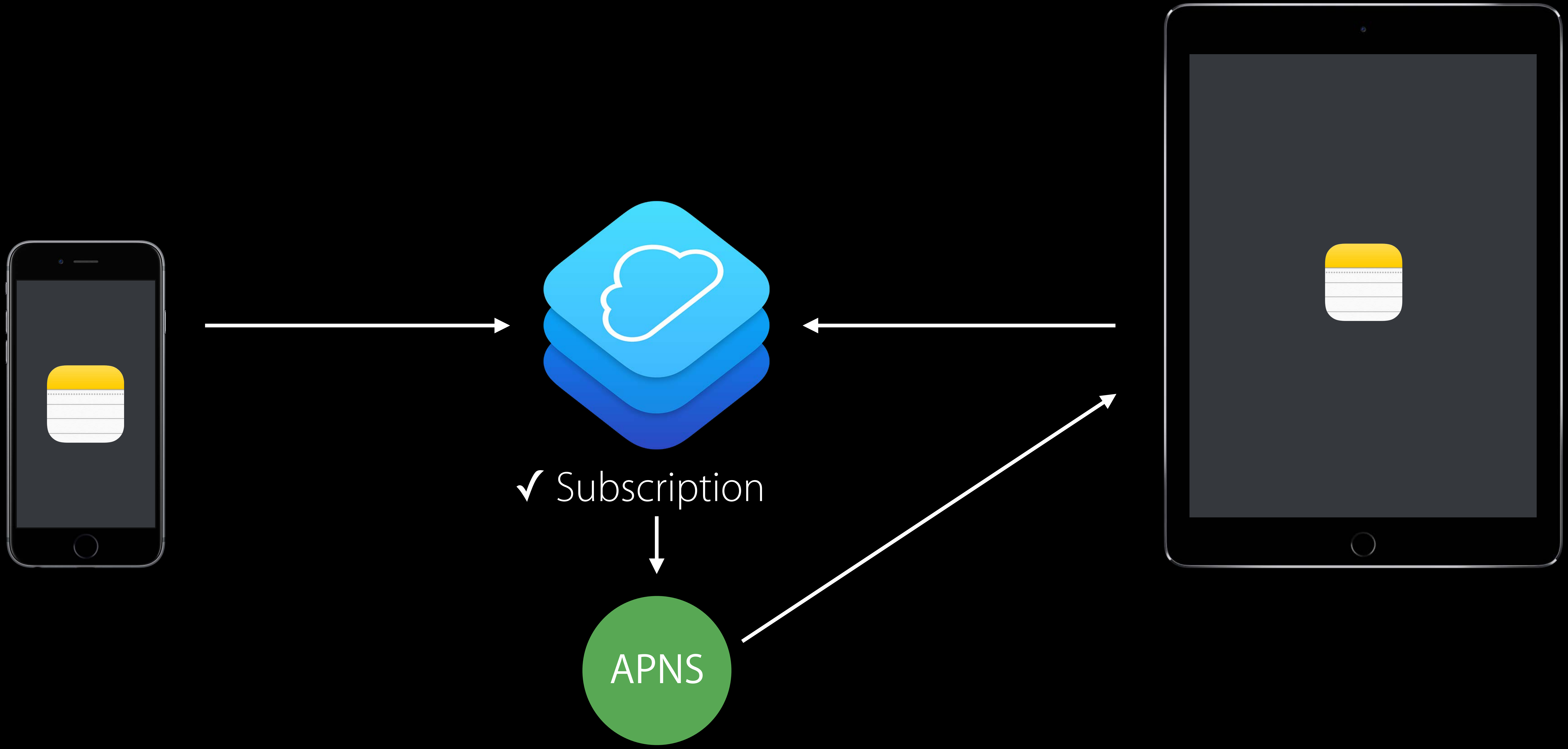
## Fetch New Changes



## Fetch New Changes



# Fetch New Changes



## Let’s Build It

## Subscribe to Changes

```
// Subscribe to changes

if (subscriptionIslocallyCached) { return }

let subscription = CKDatabaseSubscription(subscriptionID: "shared-changes")

let notificationInfo = CKNotificationInfo()
notificationInfo.shouldSendContentAvailable = true
subscription.notificationInfo = notificationInfo

let operation = CKModifySubscriptionsOperation(
   subscriptionsToSave: [subscription], subscriptionIDsToDelete: [])
operation.modifySubscriptionsCompletionBlock = { …
   if error != nil { } // Handle the error
   else { self.subscriptionIslocallyCached = true }
}

operation.qualityOfService = .utility
self.sharedDB.add(operation)
```

```
// Subscribe to changes

if (subscriptionIslocallyCached) { return }

let subscription = CKDatabaseSubscription(subscriptionID: "shared-changes")

let notificationInfo = CKNotificationInfo()
notificationInfo.shouldSendContentAvailable = true
subscription.notificationInfo = notificationInfo

let operation = CKModifySubscriptionsOperation(
   subscriptionsToSave: [subscription], subscriptionIDsToDelete: [])
operation.modifySubscriptionsCompletionBlock = { …
   if error != nil { } // Handle the error
   else { self.subscriptionIslocallyCached = true }
}

operation.qualityOfService = .utility
self.sharedDB.add(operation)
```

```
// Subscribe to changes

if (subscriptionIslocallyCached) { return }

let subscription = CKDatabaseSubscription(subscriptionID: "shared-changes")

let notificationInfo = CKNotificationInfo()
notificationInfo.shouldSendContentAvailable = true
subscription.notificationInfo = notificationInfo

let operation = CKModifySubscriptionsOperation(
   subscriptionsToSave: [subscription], subscriptionIDsToDelete: [])
operation.modifySubscriptionsCompletionBlock = { …
   if error != nil { } // Handle the error
   else { self.subscriptionIslocallyCached = true }
}

operation.qualityOfService = .utility
self.sharedDB.add(operation)
```

```swift
// Subscribe to changes

if (subscriptionIslocallyCached) { return }

let subscription = CKDatabaseSubscription(subscriptionID: "shared-changes")

let notificationInfo = CKNotificationInfo()
notificationInfo.shouldSendContentAvailable = true
subscription.notificationInfo = notificationInfo

let operation = CKModifySubscriptionsOperation(
   subscriptionsToSave: [subscription], subscriptionIDsToDelete: [])
operation.modifySubscriptionsCompletionBlock = { …
   if error != nil { } // Handle the error
   else { self.subscriptionIslocallyCached = true }
}

operation.qualityOfService = .utility
self.sharedDB.add(operation)
```

## Push Config

```swift
// Silent push

let notificationInfo = CKNotificationInfo()

// Set only this property
notificationInfo.shouldSendContentAvailable = true

// The device does NOT need to prompt for user acceptance!

// Register for notifications via:
application.registerForRemoteNotifications(…)
```

```swift
// Silent push

let notificationInfo = CKNotificationInfo()

// Set only this property
notificationInfo.shouldSendContentAvailable = true

// The device does NOT need to prompt for user acceptance!

// Register for notifications via:
application.registerForRemoteNotifications(…)
```

```
// Silent push

let notificationInfo = CKNotificationInfo()

// Set only this property
notificationInfo.shouldSendContentAvailable = true

// The device does NOT need to prompt for user acceptance!

// Register for notifications via:
application.registerForRemoteNotifications(…)
```

```swift
// Silent push

let notificationInfo = CKNotificationInfo()

// Set only this property
notificationInfo.shouldSendContentAvailable = true

// The device does NOT need to prompt for user acceptance!

// Register for notifications via:
application.registerForRemoteNotifications(…)
```

```swift
// UI push

let notificationInfo = CKNotificationInfo()

// Set any one of these three properties
notificationInfo.shouldBadge = true
notificationInfo.alertBody = NSLocalizedString("alertBody")
notificationInfo.soundName = "default"

// The device needs to prompt for user acceptance via:
application.registerUserNotificationSettings(…)

// Register for notifications via:
application.registerForRemoteNotifications(…)
```

```swift
// UI push

let notificationInfo = CKNotificationInfo()

// Set any one of these three properties
notificationInfo.shouldBadge = true
notificationInfo.alertBody = NSLocalizedString("alertBody")
notificationInfo.soundName = "default"

// The device needs to prompt for user acceptance via:
application.registerUserNotificationSettings(…)

// Register for notifications via:
application.registerForRemoteNotifications(…)
```

```swift
// UI push

let notificationInfo = CKNotificationInfo()

// Set any one of these three properties
notificationInfo.shouldBadge = true
notificationInfo.alertBody = NSLocalizedString("alertBody")
notificationInfo.soundName = "default"

// The device needs to prompt for user acceptance via:
application.registerUserNotificationSettings(…)

// Register for notifications via:
application.registerForRemoteNotifications(…)
```

```swift
// UI push

let notificationInfo = CKNotificationInfo()

// Set any one of these three properties
notificationInfo.shouldBadge = true
notificationInfo.alertBody = NSLocalizedString("alertBody")
notificationInfo.soundName = "default"

// The device needs to prompt for user acceptance via:
application.registerUserNotificationSettings(…)

// Register for notifications via:
application.registerForRemoteNotifications(…)
```

## Fine Print

Pushes can be coalesced depending on the conditions of the device

(e.g., airplane mode, poor network, low battery)

## Fine Print

CloudKit lets you ask for only what has changed!

```
// Subscribe to changes

if (subscriptionIslocallyCached) { return }

let subscription = CKDatabaseSubscription(subscriptionID: "shared-changes")

let notificationInfo = CKNotificationInfo()
notificationInfo.shouldSendContentAvailable = true
subscription.notificationInfo = notificationInfo

let operation = CKModifySubscriptionsOperation(subscriptionsToSave: [subscription],
   subscriptionIDsToDelete: [])
operation.modifySubscriptionsCompletionBlock = { …
   if error != nil { } // Handle the error
   else { self.subscriptionIslocallyCached = true }
}

operation.qualityOfService = .utility
self.sharedDB.add(operation)
```

```swift
// Subscribe to changes

if (subscriptionIslocallyCached) { return }

let subscription = CKDatabaseSubscription(subscriptionID: "shared-changes")

let notificationInfo = CKNotificationInfo()
notificationInfo.shouldSendContentAvailable = true
subscription.notificationInfo = notificationInfo

let operation = CKModifySubscriptionsOperation(subscriptionsToSave: [subscription],
   subscriptionIDsToDelete: [])
operation.modifySubscriptionsCompletionBlock = { …
   if error != nil { } // Handle the error
   else { self.subscriptionIslocallyCached = true }
}

operation.qualityOfService = .utility
self.sharedDB.add(operation)
```

```swift
// Subscribe to changes

if (subscriptionIslocallyCached) { return }

let subscription = CKDatabaseSubscription(subscriptionID: "shared-changes")

let notificationInfo = CKNotificationInfo()
notificationInfo.shouldSendContentAvailable = true
subscription.notificationInfo = notificationInfo

let operation = CKModifySubscriptionsOperation(subscriptionsToSave: [subscription],
   subscriptionIDsToDelete: [])
operation.modifySubscriptionsCompletionBlock = { …
   if error != nil { } // Handle the error
   else { self.subscriptionIslocallyCached = true }
}

operation.qualityOfService = .utility
self.sharedDB.add(operation)
```

```swift
// Subscribe to changes

if (subscriptionIslocallyCached) { return }

let subscription = CKDatabaseSubscription(subscriptionID: "shared-changes")

let notificationInfo = CKNotificationInfo()
notificationInfo.shouldSendContentAvailable = true
subscription.notificationInfo = notificationInfo

let operation = CKModifySubscriptionsOperation(subscriptionsToSave: [subscription],
   subscriptionIDsToDelete: [])
operation.modifySubscriptionsCompletionBlock = { …
   if error != nil { } // Handle the error
   else { self.subscriptionIslocallyCached = true }
}

operation.qualityOfService = .utility
self.sharedDB.add(operation)
```

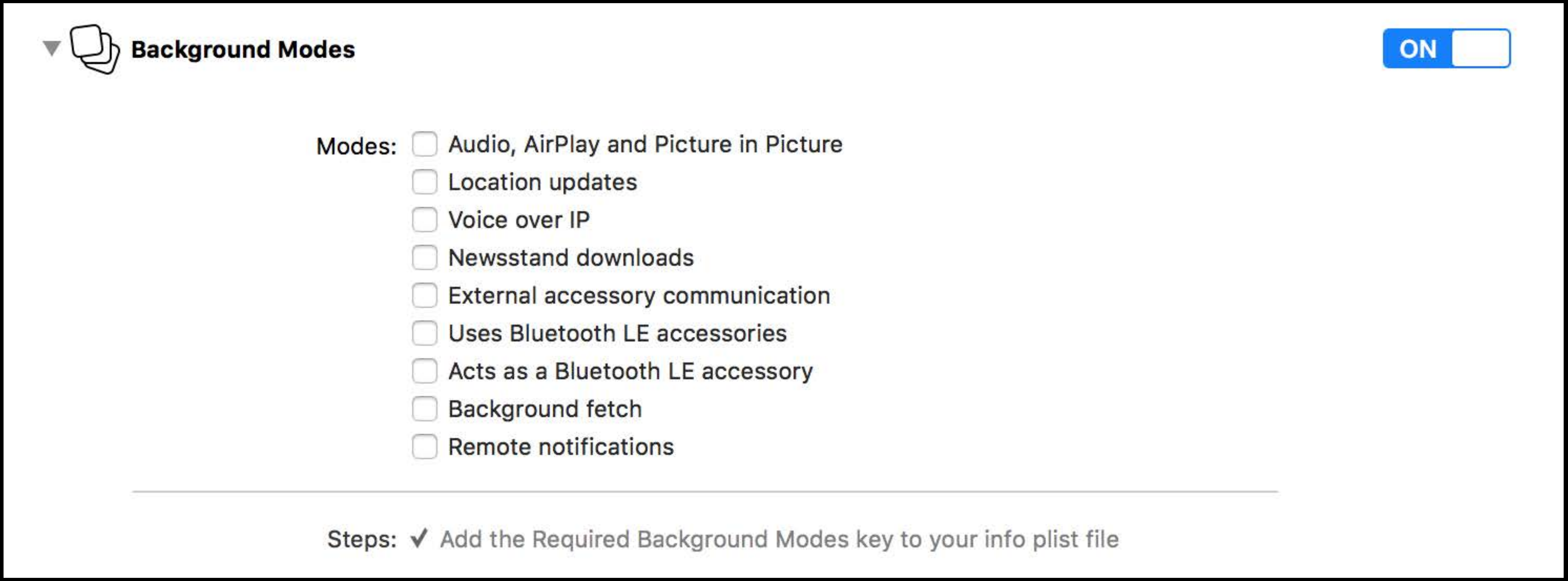
```
// Subscribe to changes

if (subscriptionIslocallyCached) { return }

let subscription = CKDatabaseSubscription(subscriptionID: "shared-changes")

let notificationInfo = CKNotificationInfo()
notificationInfo.shouldSendContentAvailable = true
subscription.notificationInfo = notificationInfo

let operation = CKModifySubscriptionsOperation(subscriptionsToSave: [subscription],
   subscriptionIDsToDelete: [])
operation.modifySubscriptionsCompletionBlock = { …
   if error != nil { } // Handle the error
   else { self.subscriptionIslocallyCached = true }
}

operation.qualityOfService = .utility
self.sharedDB.add(operation)
```

```swift
// Subscribe to changes

if (subscriptionIslocallyCached) { return }

let subscription = CKDatabaseSubscription(subscriptionID: "shared-changes")

let notificationInfo = CKNotificationInfo()
notificationInfo.shouldSendContentAvailable = true
subscription.notificationInfo = notificationInfo

let operation = CKModifySubscriptionsOperation(subscriptionsToSave: [subscription],
   subscriptionIDsToDelete: [])
operation.modifySubscriptionsCompletionBlock = { …
   if error != nil { } // Handle the error
   else { self.subscriptionIslocallyCached = true }
}

operation.qualityOfService = .utility
self.sharedDB.add(operation)
```

```swift
// Subscribe to changes

if (subscriptionIslocallyCached) { return }

let subscription = CKDatabaseSubscription(subscriptionID: "shared-changes")

let notificationInfo = CKNotificationInfo()
notificationInfo.shouldSendContentAvailable = true
subscription.notificationInfo = notificationInfo

let operation = CKModifySubscriptionsOperation(subscriptionsToSave: [subscription],
   subscriptionIDsToDelete: [])
operation.modifySubscriptionsCompletionBlock = { …
   if error != nil { } // Handle the error
   else { self.subscriptionIslocallyCached = true }
}

operation.qualityOfService = .utility
self.sharedDB.add(operation)
```

✓ Subscribed to changes

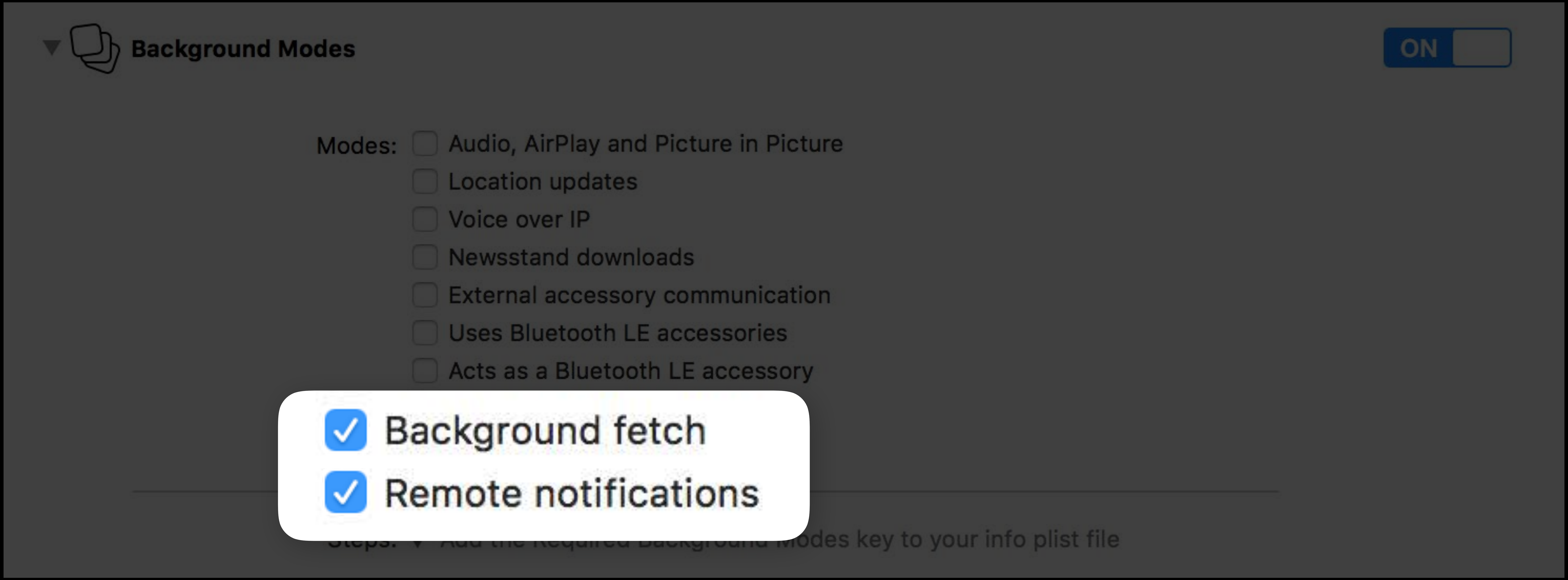
## Listen for Pushes


Background Modes

ON

Modes:
- [ ] Audio, AirPlay and Picture in Picture
- [ ] Location updates
- [ ] Voice over IP
- [ ] Newsstand downloads
- [ ] External accessory communication
- [ ] Uses Bluetooth LE accessories
- [ ] Acts as a Bluetooth LE accessory
- [ ] Background fetch
- [ ] Remote notifications

Steps: ✓ Add the Required Background Modes key to your info plist file



Background Modes ON

Modes:
- [ ] Audio, AirPlay and Picture in Picture
- [ ] Location updates
- [ ] Voice over IP
- [ ] Newsstand downloads
- [ ] External accessory communication
- [ ] Uses Bluetooth LE accessories
- [ ] Acts as a Bluetooth LE accessory
- [✓] Background fetch
- [✓] Remote notifications

Steps: ✓ Add the Required Background Modes key to your info plist file

```
// Listen for pushes

func application(_ application: UIApplication,
   didReceiveRemoteNotification userInfo: [NSObject : AnyObject],
   fetchCompletionHandler completionHandler: (UIBackgroundFetchResult) -> Void) {

   let dict = userInfo as! [String: NSObject]
   let notification = CKNotification(fromRemoteNotificationDictionary: dict)

   if (notification.subscriptionID == "shared-changes") {
      fetchSharedChanges {
         completionHandler(UIBackgroundFetchResult.newData)
      }
   }
}
```

```swift
// Listen for pushes

func application(_ application: UIApplication,
   didReceiveRemoteNotification userInfo: [NSObject : AnyObject],
   fetchCompletionHandler completionHandler: (UIBackgroundFetchResult) -> Void) {

   let dict = userInfo as! [String: NSObject]
   let notification = CKNotification(fromRemoteNotificationDictionary: dict)

   if (notification.subscriptionID == "shared-changes") {
      fetchSharedChanges {
         completionHandler(UIBackgroundFetchResult.newData)
      }
   }
}
```

```swift
// Listen for pushes

func application(_ application: UIApplication,
   didReceiveRemoteNotification userInfo: [NSObject : AnyObject],
   fetchCompletionHandler completionHandler: (UIBackgroundFetchResult) -> Void) {

   let dict = userInfo as! [String: NSObject]
   let notification = CKNotification(fromRemoteNotificationDictionary: dict)

   if (notification.subscriptionID == "shared-changes") {
      fetchSharedChanges {
         completionHandler(UIBackgroundFetchResult.newData)
      }
   }
}
```

```swift
// Listen for pushes

func application(_ application: UIApplication,
   didReceiveRemoteNotification userInfo: [NSObject : AnyObject],
   fetchCompletionHandler completionHandler: (UIBackgroundFetchResult) -> Void) {

   let dict = userInfo as! [String: NSObject]
   let notification = CKNotification(fromRemoteNotificationDictionary: dict)

   if (notification.subscriptionID == "shared-changes") {
      fetchSharedChanges {
         completionHandler(UIBackgroundFetchResult.newData)
      }
   }
}
```

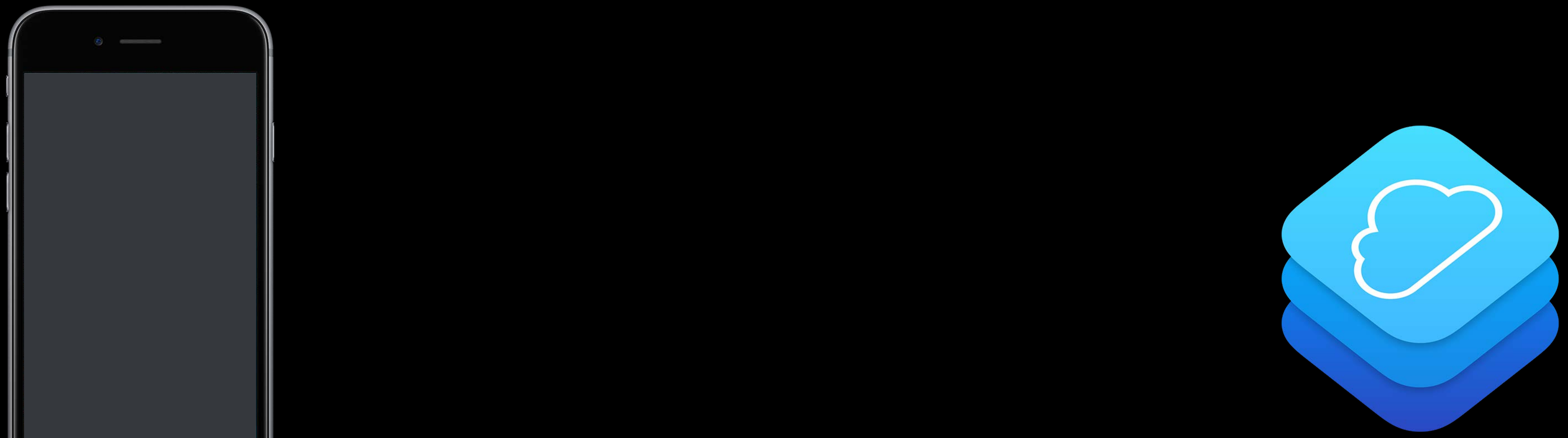
```swift
// Listen for pushes

func application(_ application: UIApplication,
   didReceiveRemoteNotification userInfo: [NSObject : AnyObject],
   fetchCompletionHandler completionHandler: (UIBackgroundFetchResult) -> Void) {

   let dict = userInfo as! [String: NSObject]
   let notification = CKNotification(fromRemoteNotificationDictionary: dict)

   if (notification.subscriptionID == "shared-changes") {
      fetchSharedChanges {
         completionHandler(UIBackgroundFetchResult.newData)
      }
   }
}
```

✓ Listening for pushes

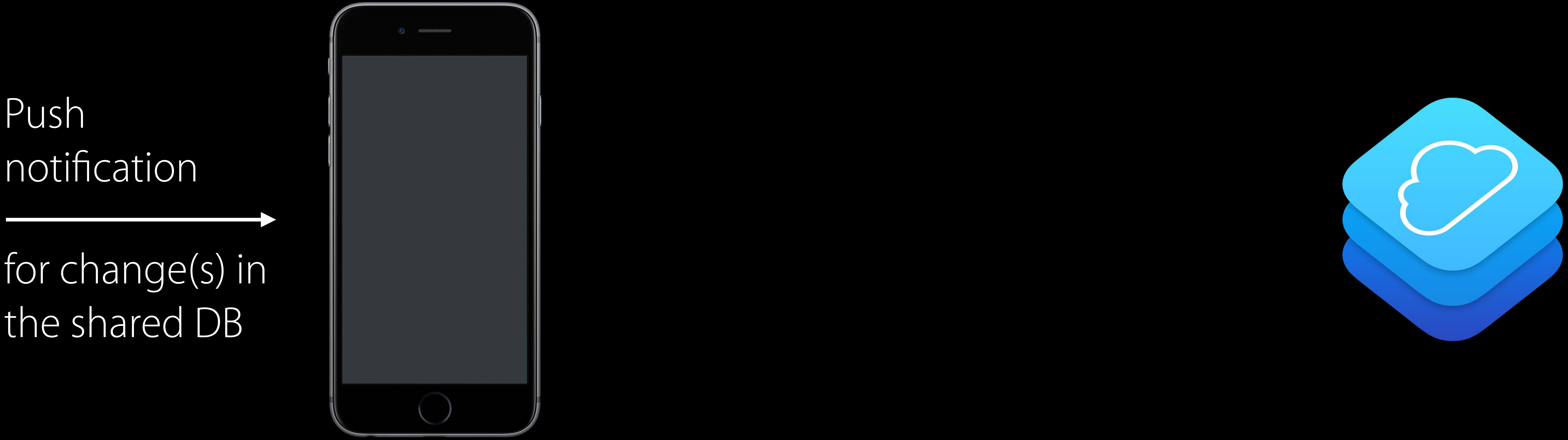
## Some Time Later, We Get a Push…

## Fetch New Changes

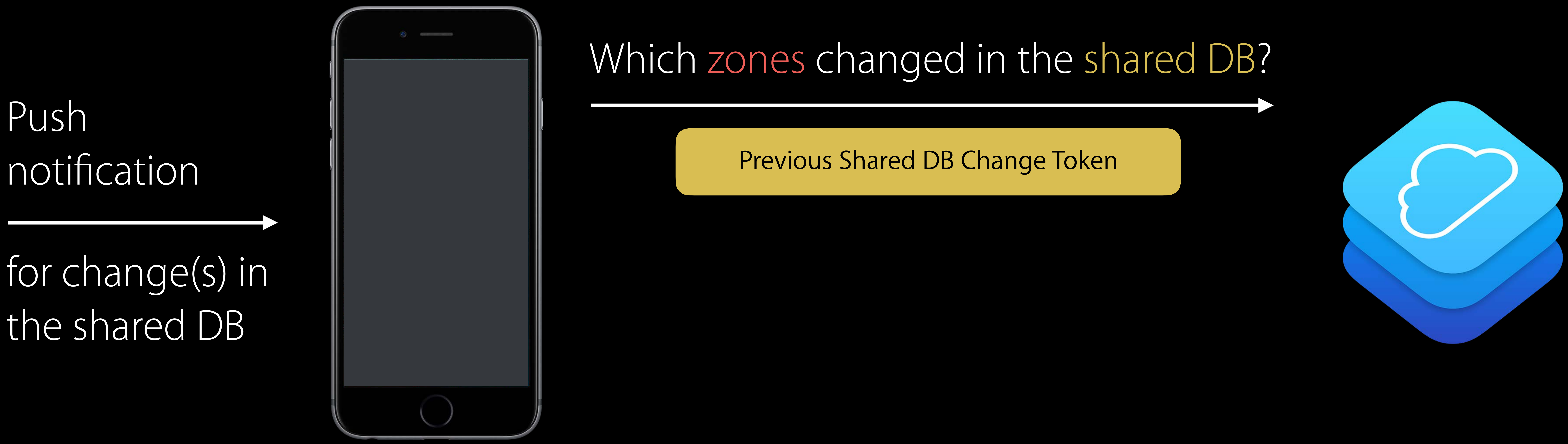
## Fetching Changes



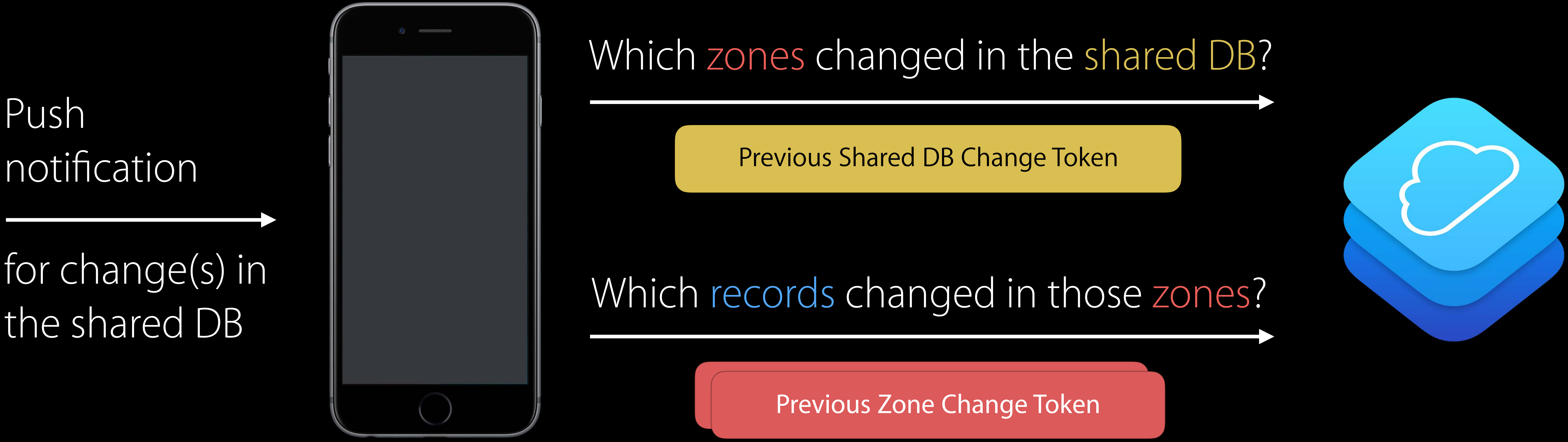
# Fetching Changes

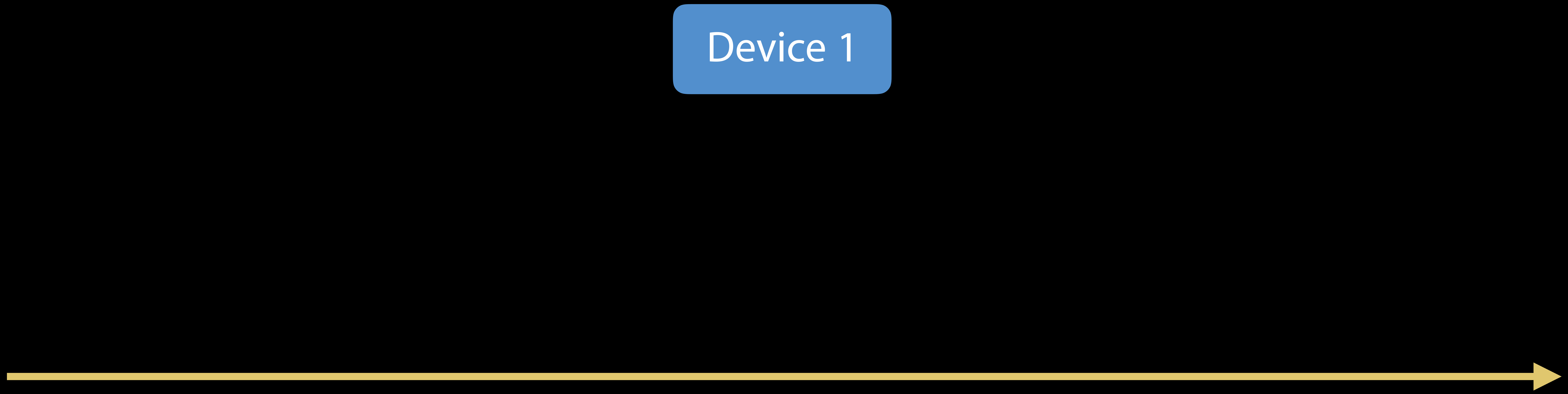


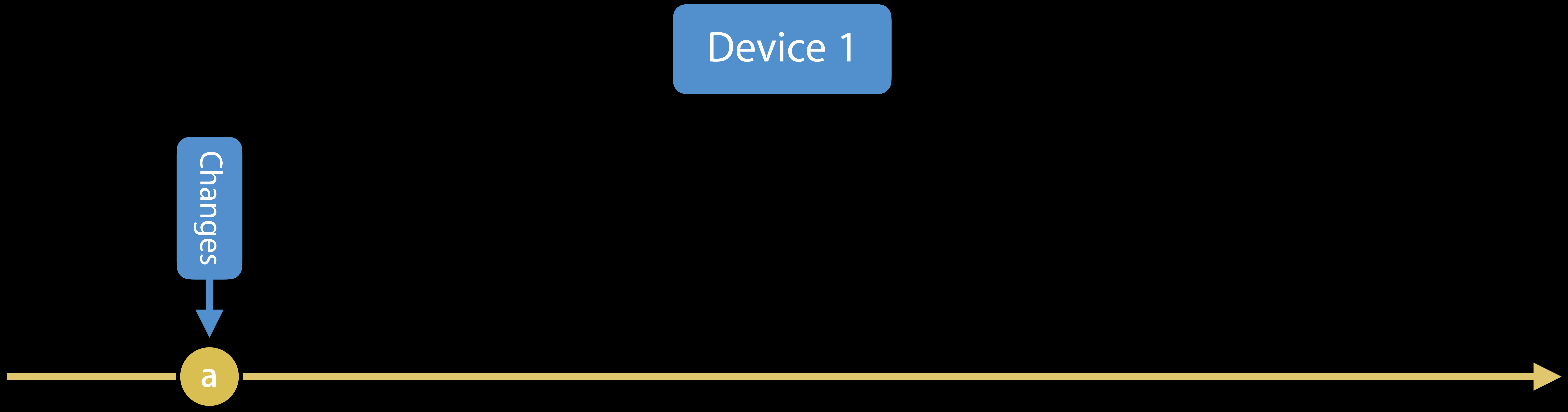
# Fetching Changes

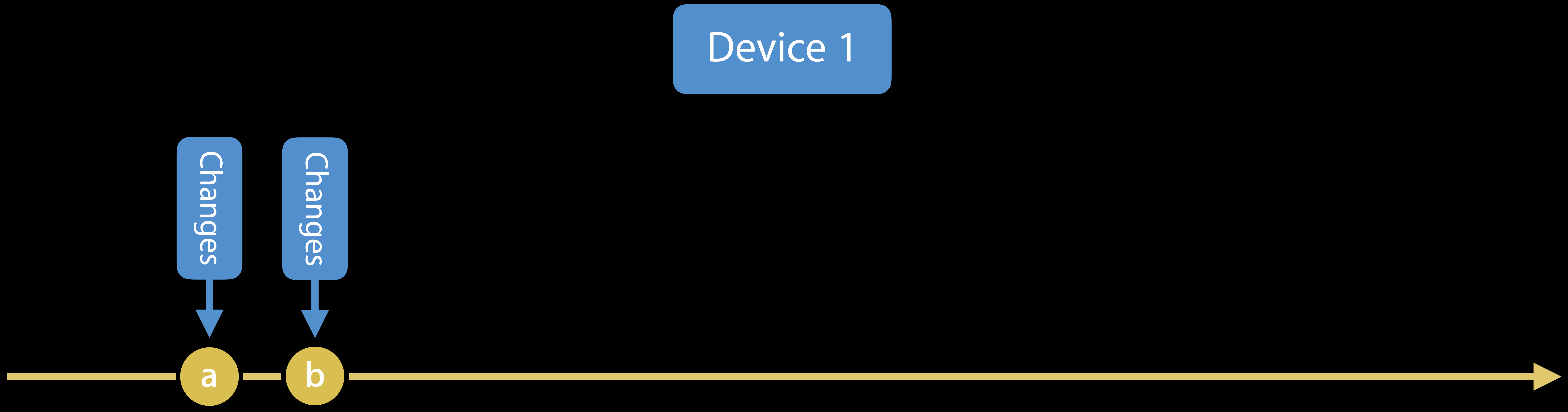

Push notification
for change(s) in the shared DB
Which zones changed in the shared DB?
Previous Shared DB Change Token

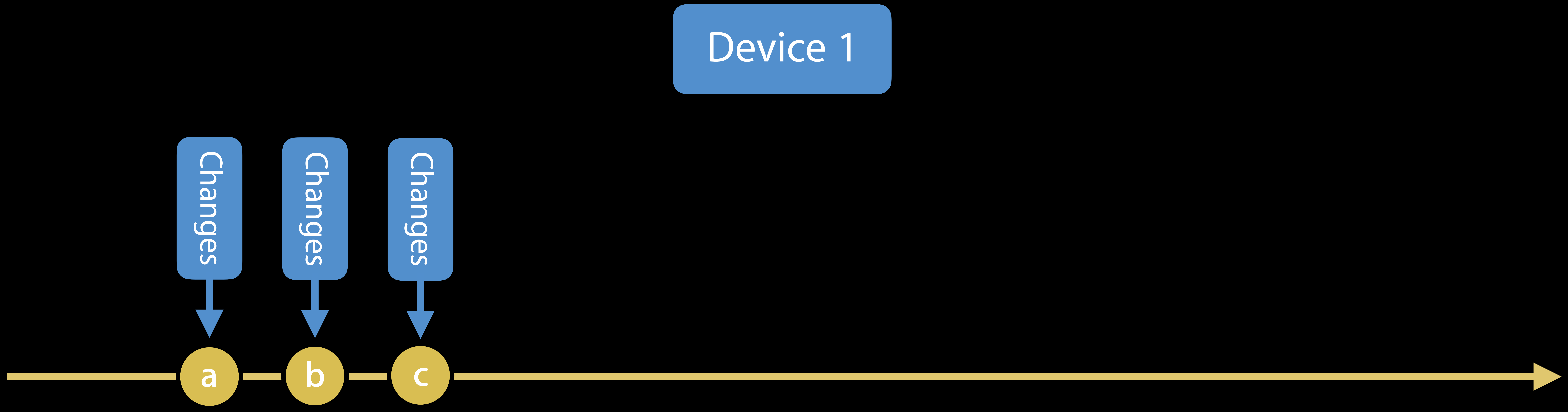
# Fetching Changes

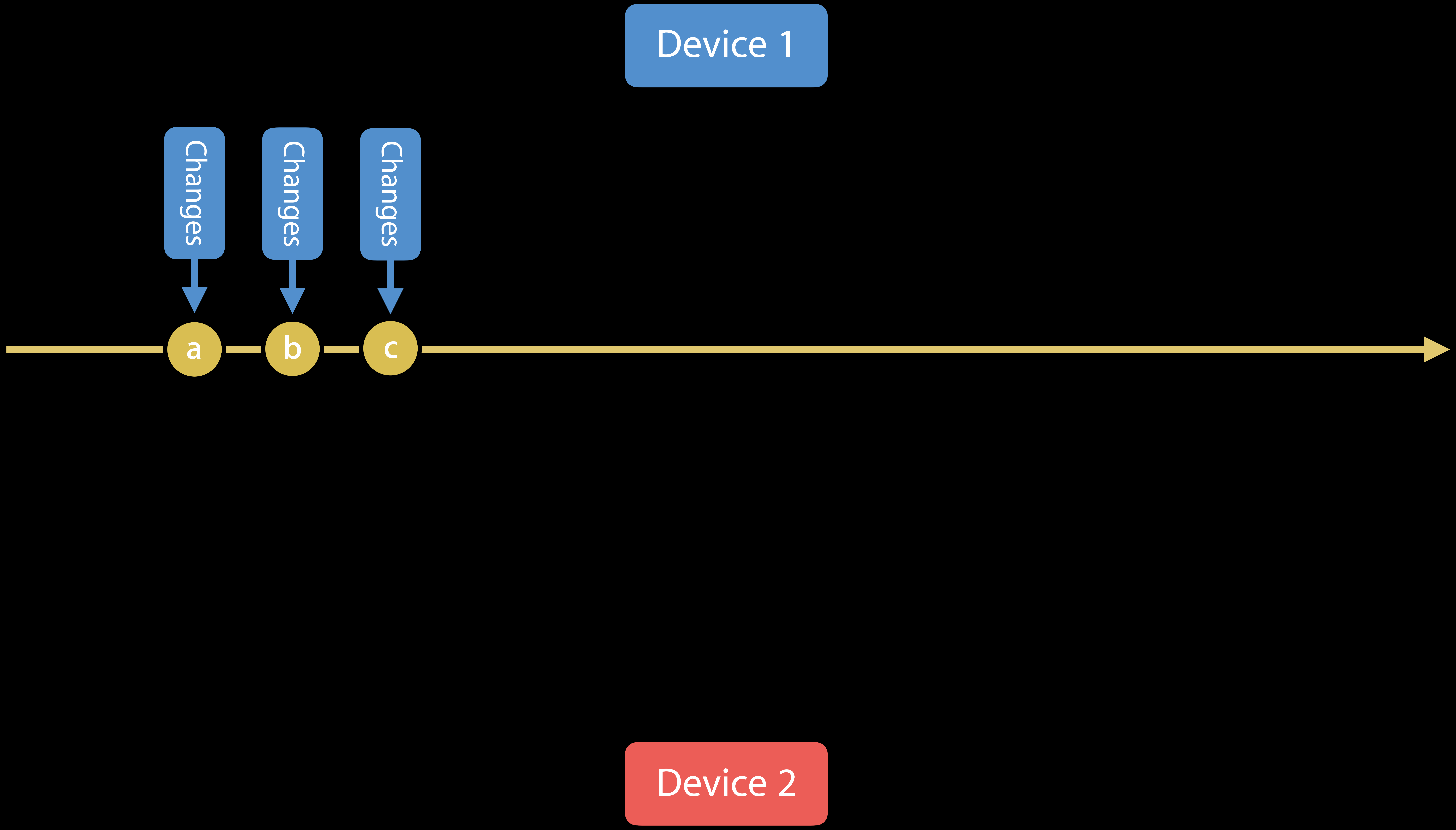

Push notification
for change(s) in the shared DB
Which zones changed in the shared DB?
Previous Shared DB Change Token
Which records changed in those zones?
Previous Zone Change Token

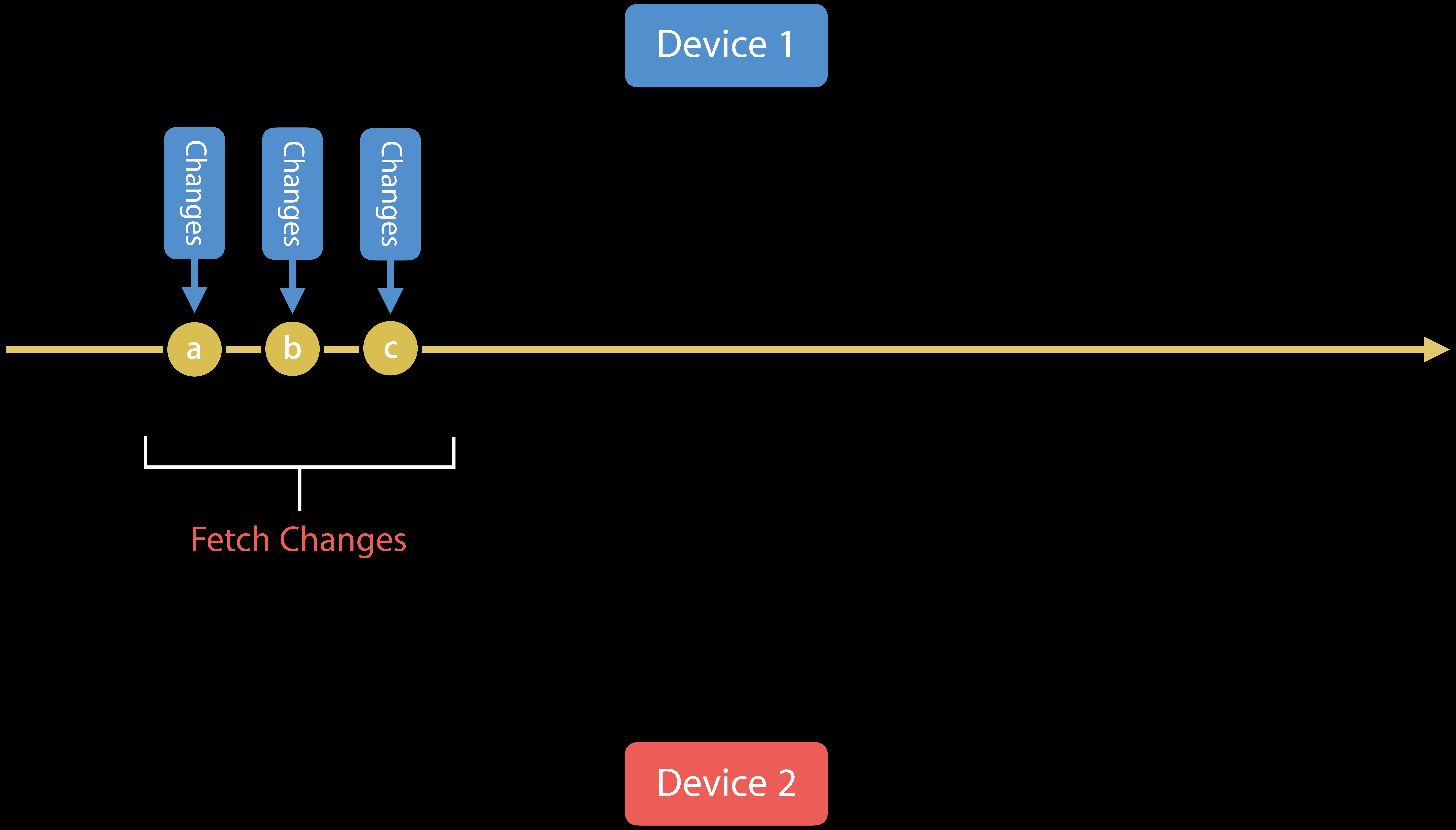

Device 1

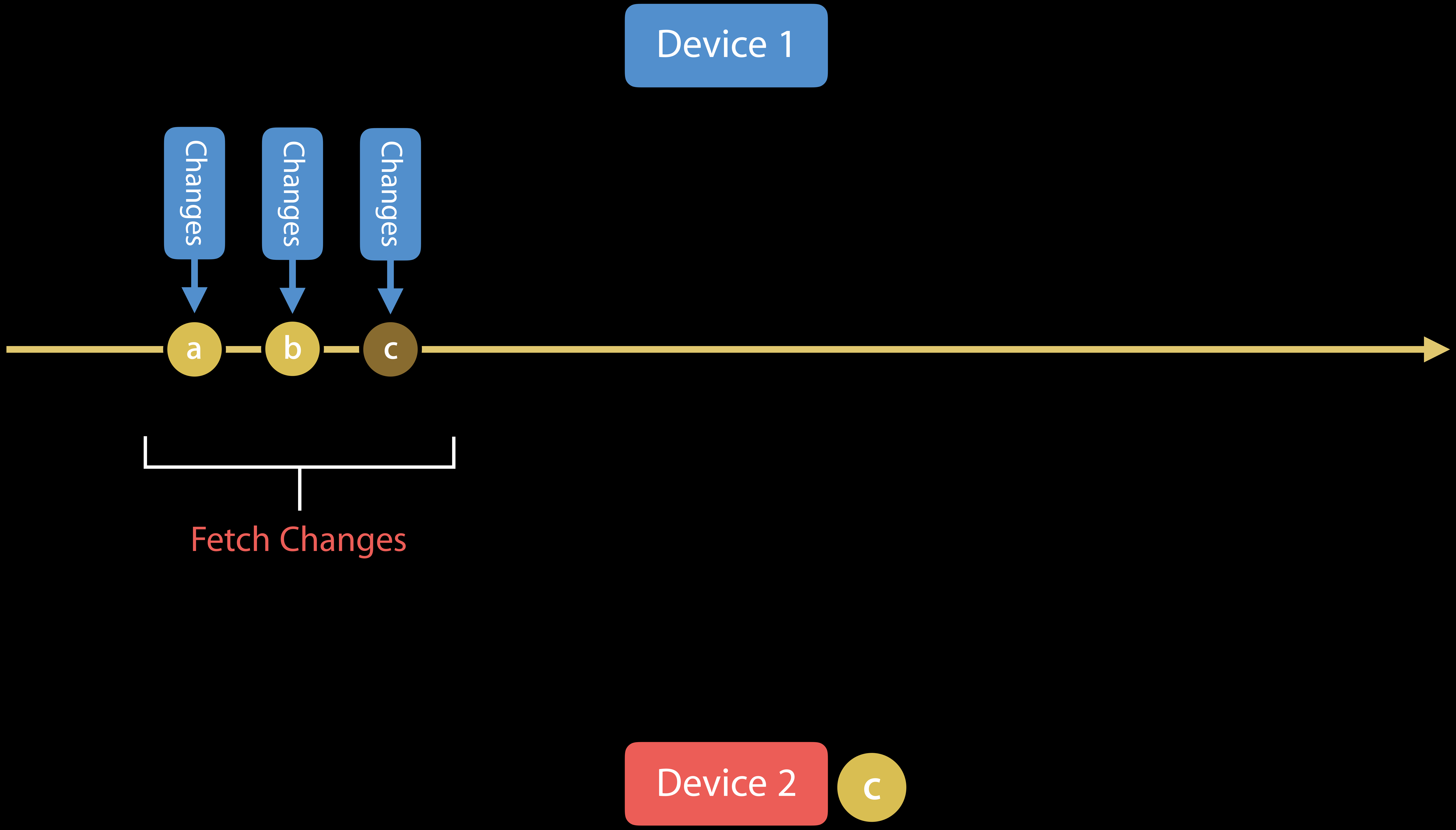

Device 1
Changes
a

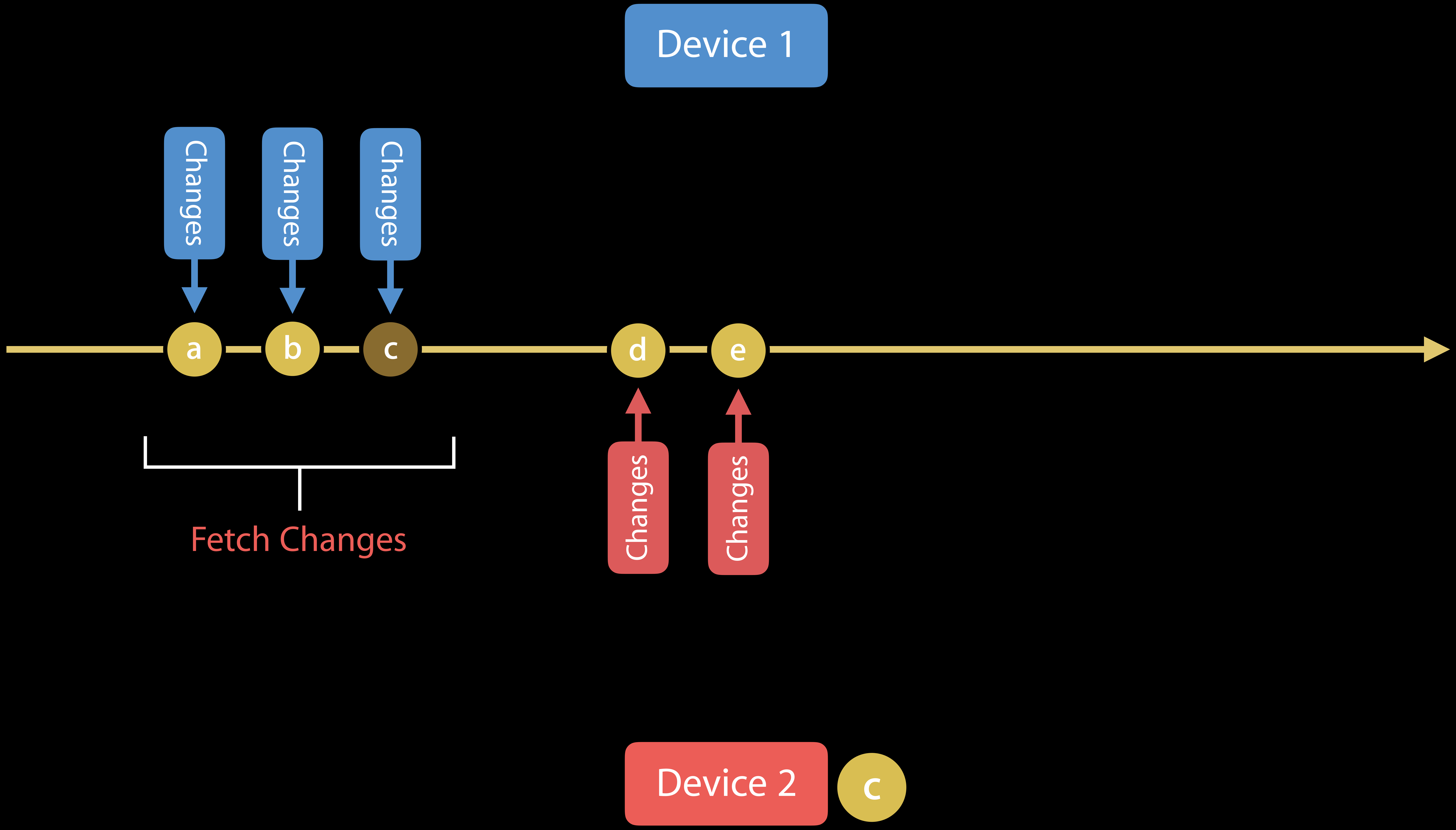

Device 1
Changes
Changes
a
b

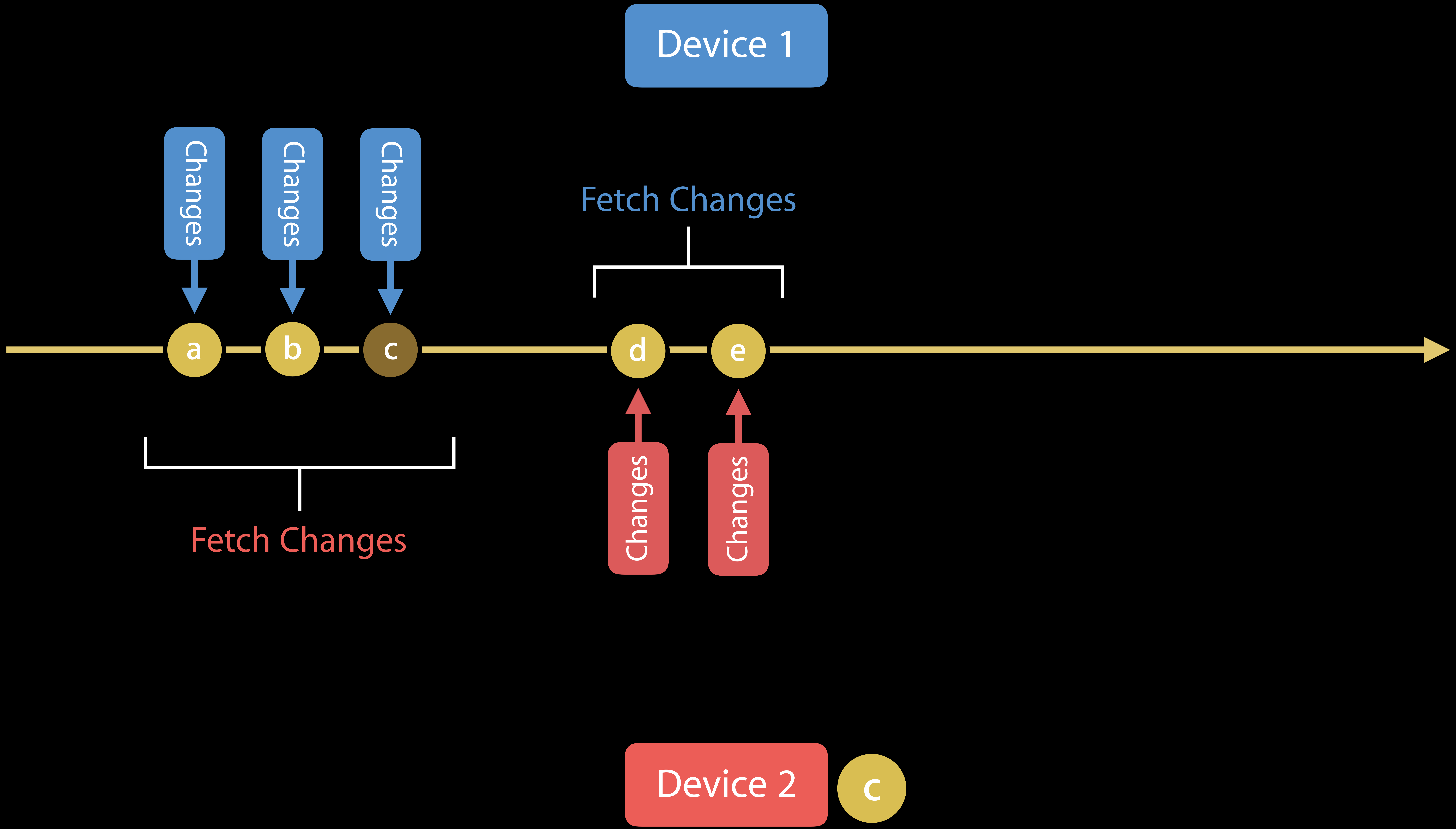

Device 1
Changes
Changes
Changes
a
b
c

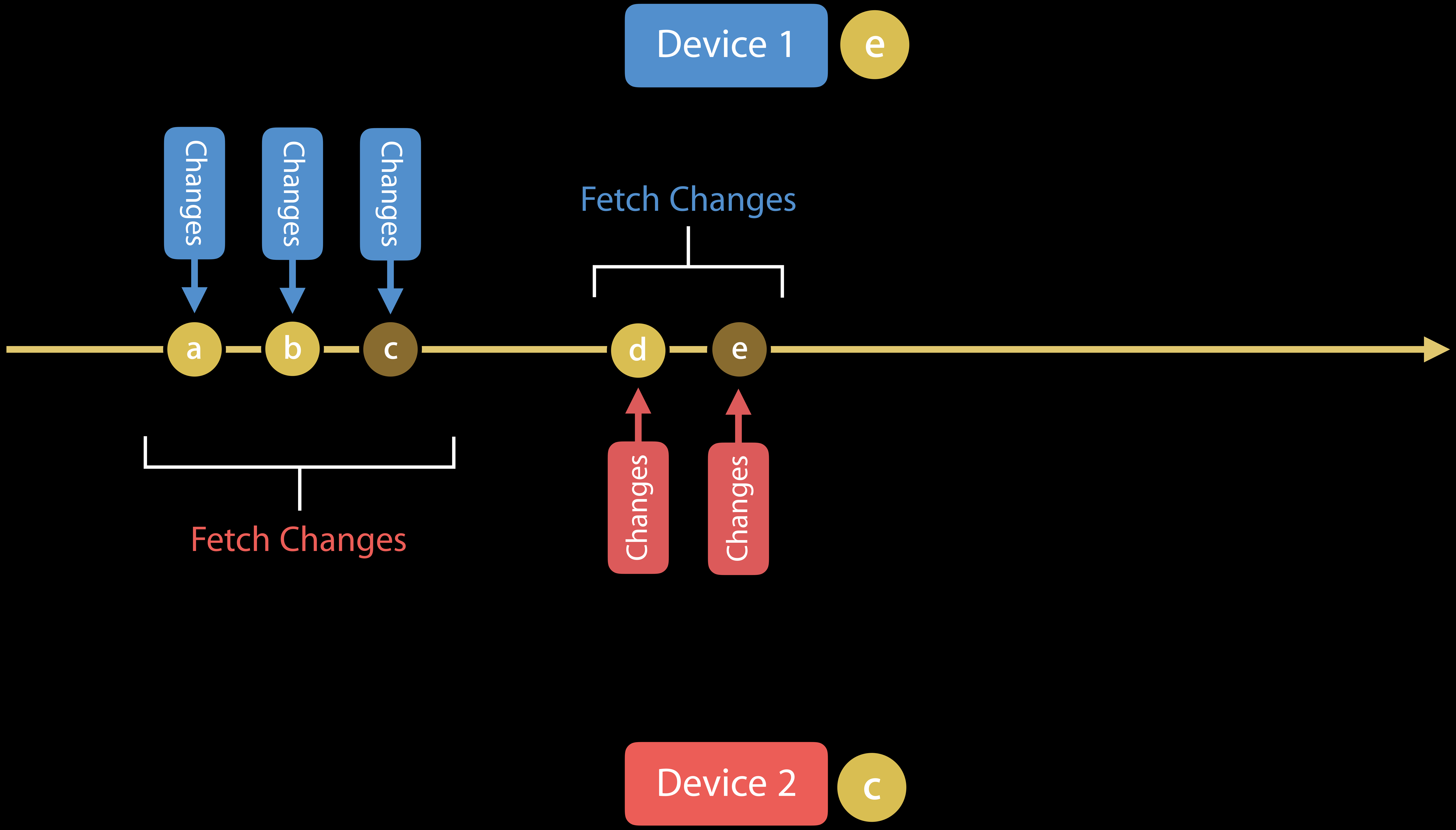

Device 1
Changes
Changes
Changes
a
b
c
Device 2

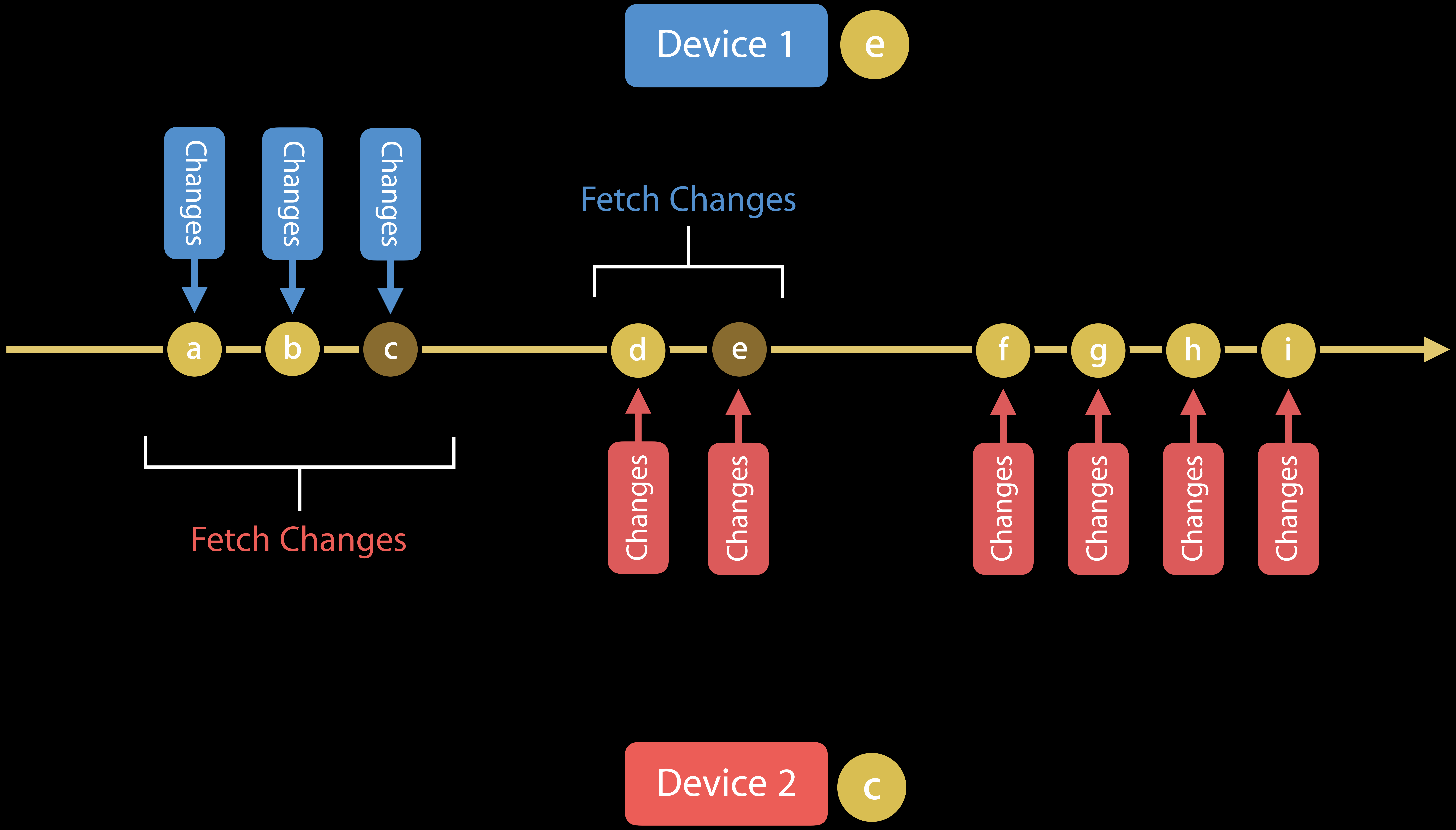

Device 1
Changes
Changes
Changes
a
b
c
Fetch Changes
Device 2

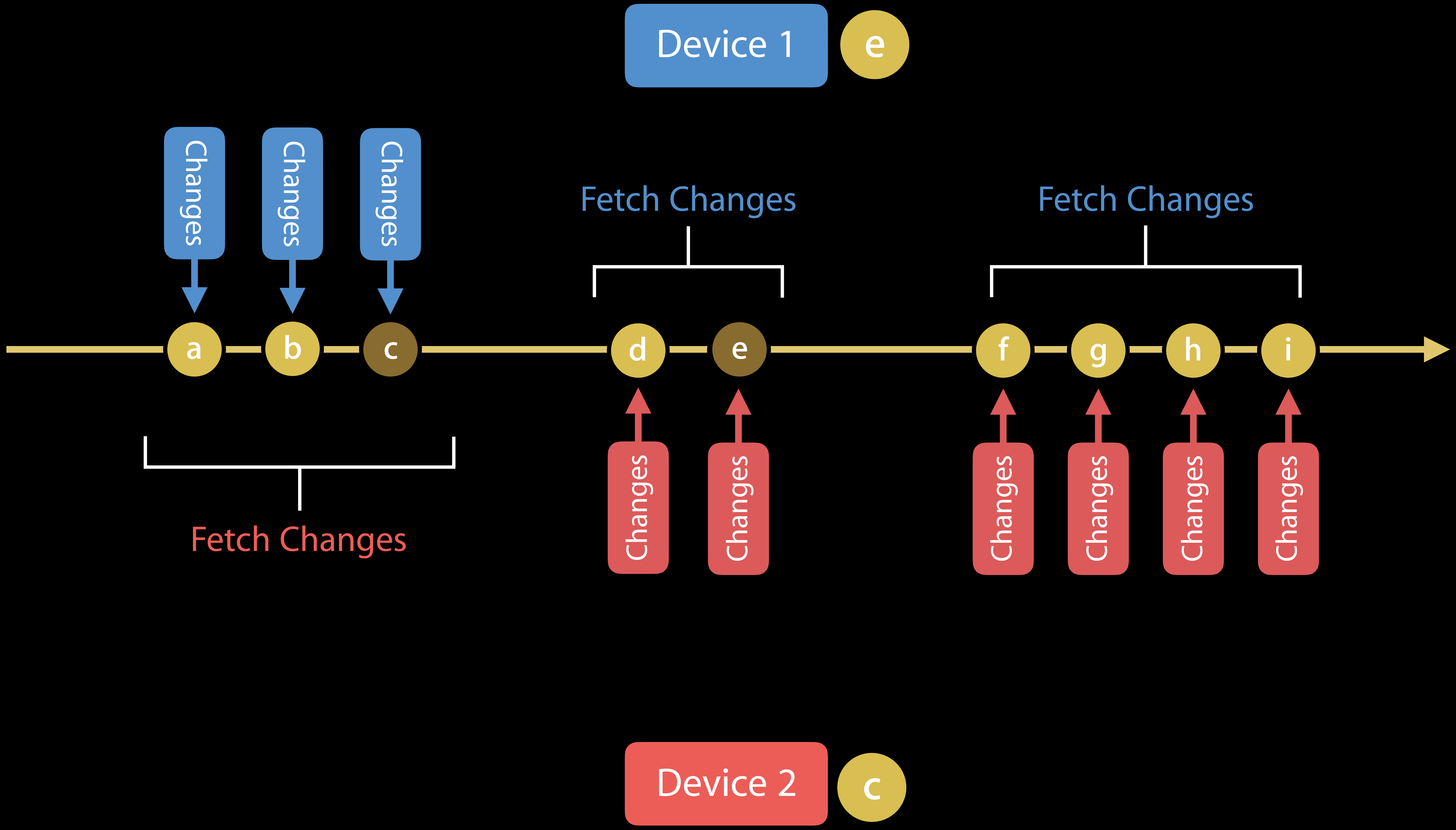

Device 1
Changes
Changes
Changes
a
b
c
Fetch Changes
Device 2
c

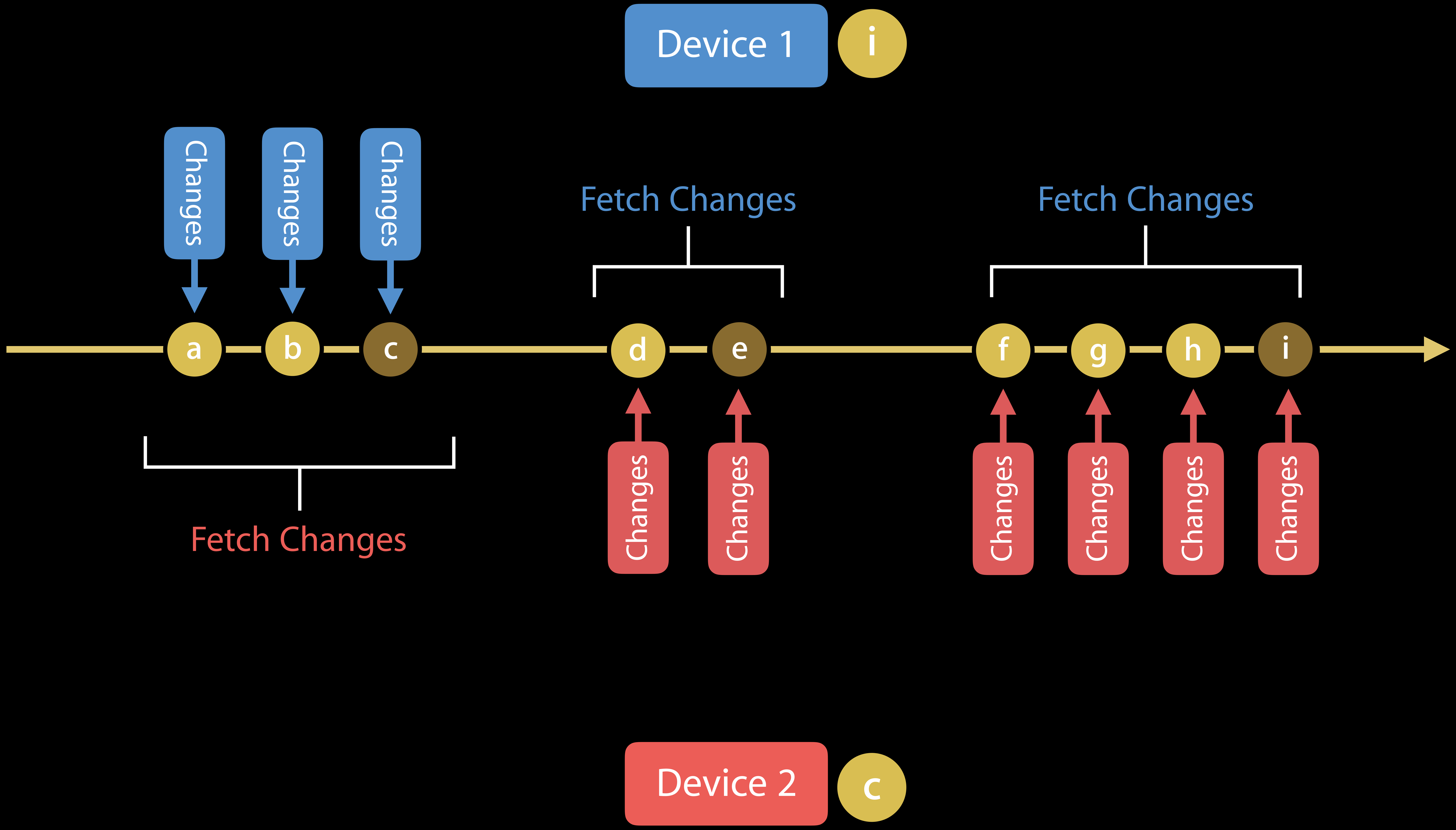

Device 1
Changes
Changes
Changes
a
b
c
d
e
Fetch Changes
Changes
Changes
Device 2
c

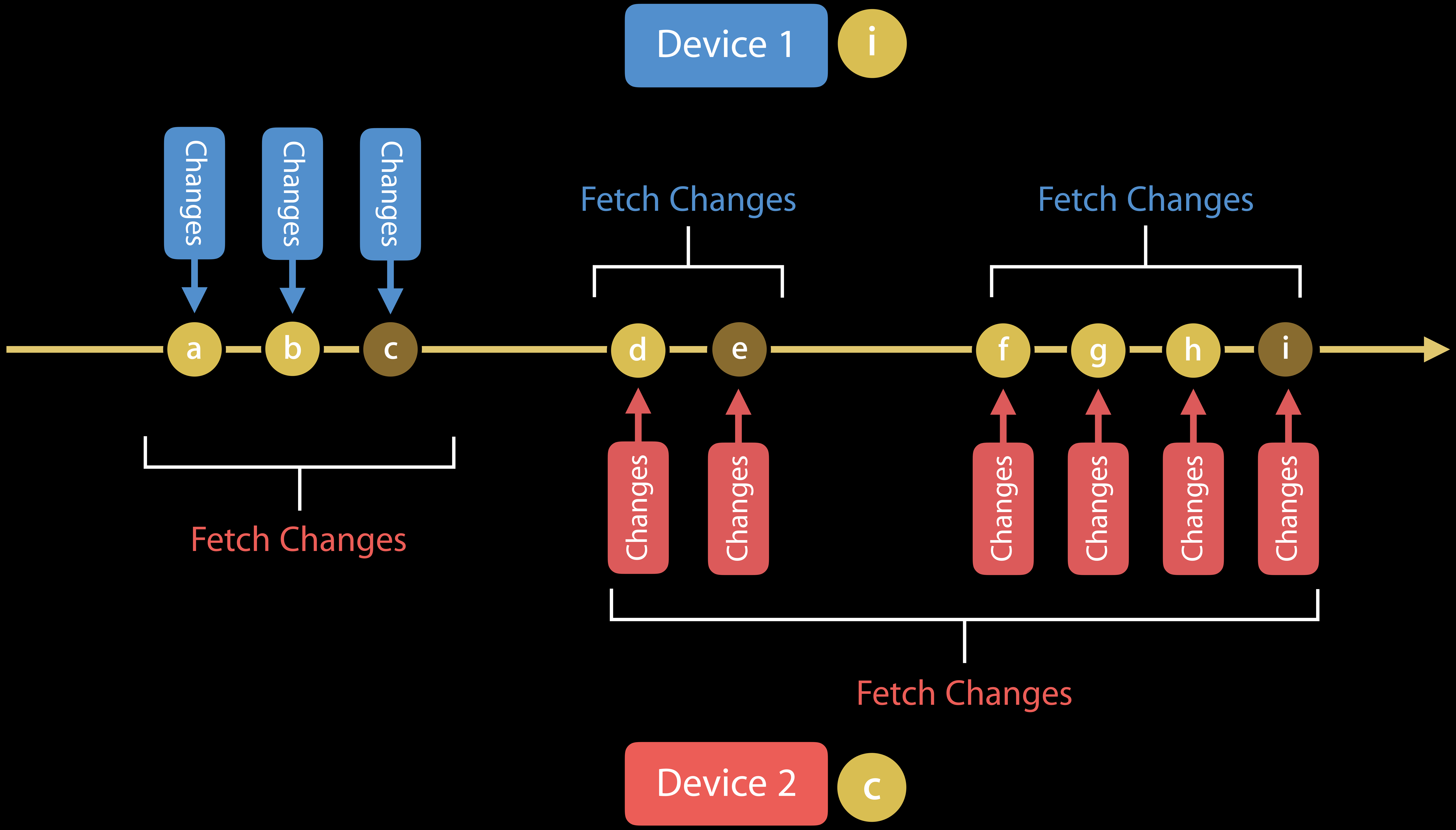

Device 1
Changes
Changes
Changes
Fetch Changes
a
b
c
d
e
Fetch Changes
Changes
Changes
Device 2
c

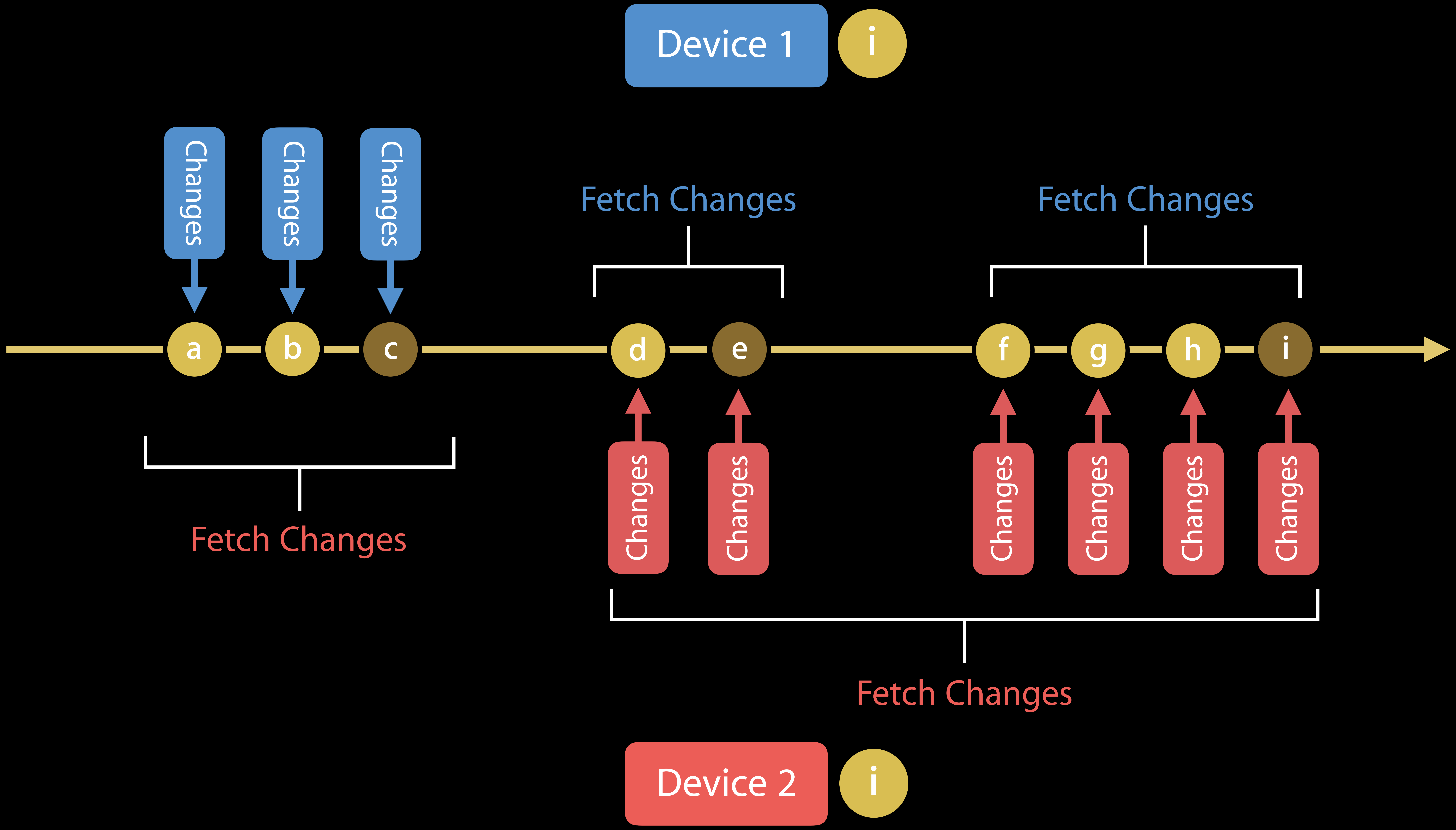

Device 1
e
Changes
Changes
Changes
Fetch Changes
a
b
c
d
e
Fetch Changes
Changes
Changes
Device 2
c


Device 1
e
Changes
Changes
Changes
Fetch Changes
a
b
c
d
e
f
g
h
i
Fetch Changes
Changes
Changes
Changes
Changes
Changes
Changes
Device 2
c


Device 1
e
Changes
Changes
Changes
Fetch Changes
Fetch Changes
a
b
c
d
e
f
g
h
i
Fetch Changes
Changes
Changes
Changes
Changes
Changes
Changes
Device 2
c


Device 1
i
Changes
Changes
Changes
Fetch Changes
Fetch Changes
a
b
c
d
e
f
g
h
i
Fetch Changes
Changes
Changes
Changes
Changes
Changes
Changes
Device 2
c


Device 1
i
Changes
Changes
Changes
Fetch Changes
Fetch Changes
a
b
c
d
e
f
g
h
i
Fetch Changes
Changes
Changes
Changes
Changes
Changes
Changes
Fetch Changes
Device 2
c


Device 1
i
Changes
Changes
Changes
Fetch Changes
Fetch Changes
a
b
c
d
e
f
g
h
i
Fetch Changes
Changes
Changes
Changes
Changes
Changes
Changes
Fetch Changes
Device 2
i

```
// Fetching database changes after a push

func fetchSharedChanges(_ callback: () -> Void) {
   let changesOperation = CKFetchDatabaseChangesOperation(
      previousServerChangeToken: sharedDBChangeToken) // previously cached

   changesOperation.fetchAllChanges = true
   changesOperation.recordZoneWithIDChangedBlock = { … } // collect zone IDs
   changesOperation.recordZoneWithIDWasDeletedBlock = { … } // delete local cache
   changesOperation.changeTokenUpdatedBlock = { … } // cache new token

   changesOperation.fetchDatabaseChangesCompletionBlock = {
      (newToken: CKServerChangeToken?, more: Bool, error: NSError?) -> Void in
      // error handling here
      self.sharedDBChangeToken = newToken // cache new token
      self.fetchZoneChanges(callback) // using CKFetchRecordZoneChangesOperation
   }
   self.sharedDB.add(changesOperation)
}
```

```
// Fetching database changes after a push

func fetchSharedChanges(_ callback: () -> Void) {
    let changesOperation = CKFetchDatabaseChangesOperation(
        previousServerChangeToken: sharedDBChangeToken) // previously cached

    changesOperation.fetchAllChanges = true
    changesOperation.recordZoneWithIDChangedBlock = { … } // collect zone IDs
    changesOperation.recordZoneWithIDWasDeletedBlock = { … } // delete local cache
    changesOperation.changeTokenUpdatedBlock = { … } // cache new token

    changesOperation.fetchDatabaseChangesCompletionBlock = {
        (newToken: CKServerChangeToken?, more: Bool, error: NSError?) -> Void in
        // error handling here
        self.sharedDBChangeToken = newToken // cache new token
        self.fetchZoneChanges(callback) // using CKFetchRecordZoneChangesOperation
    }
    self.sharedDB.add(changesOperation)
}
```

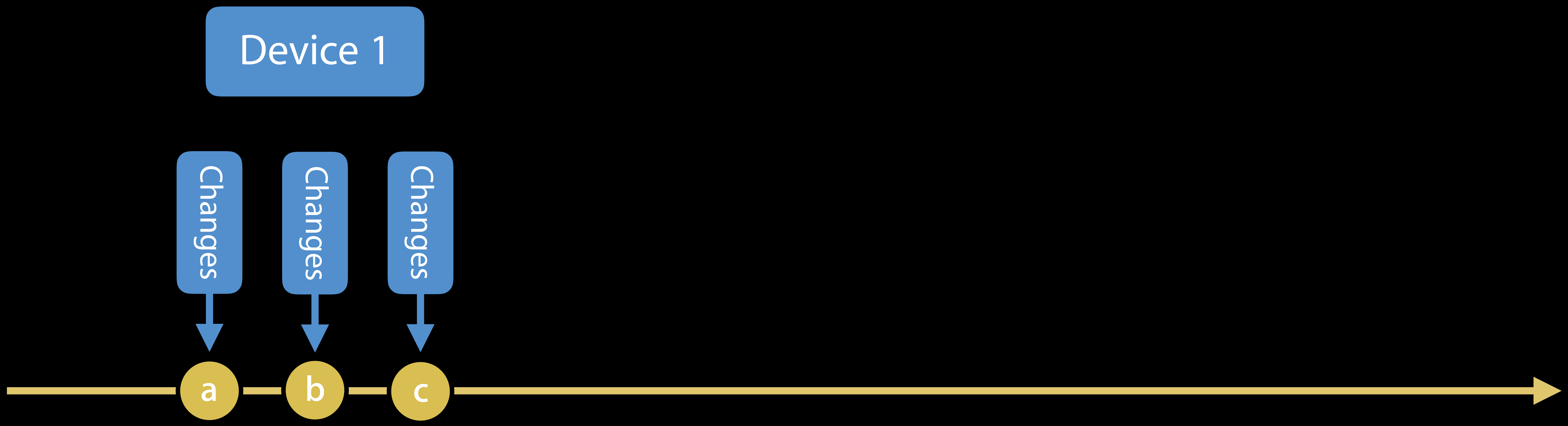
```
// Fetching database changes after a push

func fetchSharedChanges(_ callback: () -> Void) {
    let changesOperation = CKFetchDatabaseChangesOperation(
        previousServerChangeToken: sharedDBChangeToken) // previously cached

    changesOperation.fetchAllChanges = true
    changesOperation.recordZoneWithIDChangedBlock = { … } // collect zone IDs
    changesOperation.recordZoneWithIDWasDeletedBlock = { … } // delete local cache
    changesOperation.changeTokenUpdatedBlock = { … } // cache new token

    changesOperation.fetchDatabaseChangesCompletionBlock = {
        (newToken: CKServerChangeToken?, more: Bool, error: NSError?) -> Void in
        // error handling here
        self.sharedDBChangeToken = newToken // cache new token
        self.fetchZoneChanges(callback) // using CKFetchRecordZoneChangesOperation
    }
    self.sharedDB.add(changesOperation)
}
```

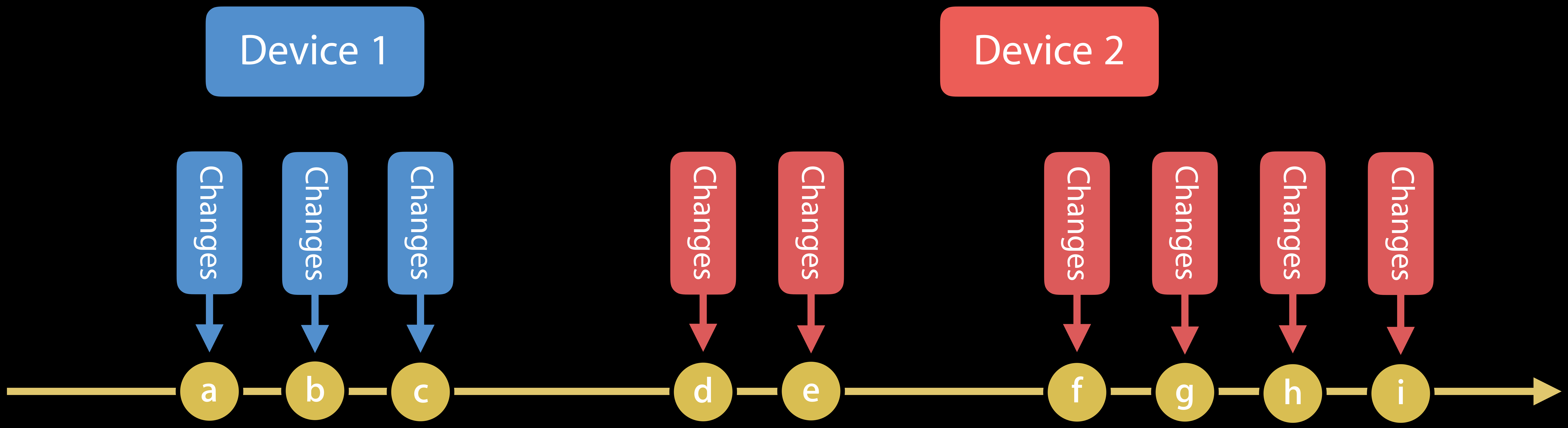
```swift
// Fetching database changes after a push

func fetchSharedChanges(_ callback: () -> Void) {
   let changesOperation = CKFetchDatabaseChangesOperation(
      previousServerChangeToken: sharedDBChangeToken) // previously cached

   changesOperation.fetchAllChanges = true
   changesOperation.recordZoneWithIDChangedBlock = { … } // collect zone IDs
   changesOperation.recordZoneWithIDWasDeletedBlock = { … } // delete local cache
   changesOperation.changeTokenUpdatedBlock = { … } // cache new token

   changesOperation.fetchDatabaseChangesCompletionBlock = {
      (newToken: CKServerChangeToken?, more: Bool, error: NSError?) -> Void in
      // error handling here
      self.sharedDBChangeToken = newToken // cache new token
      self.fetchZoneChanges(callback) // using CKFetchRecordZoneChangesOperation
   }
   self.sharedDB.add(changesOperation)
}
```

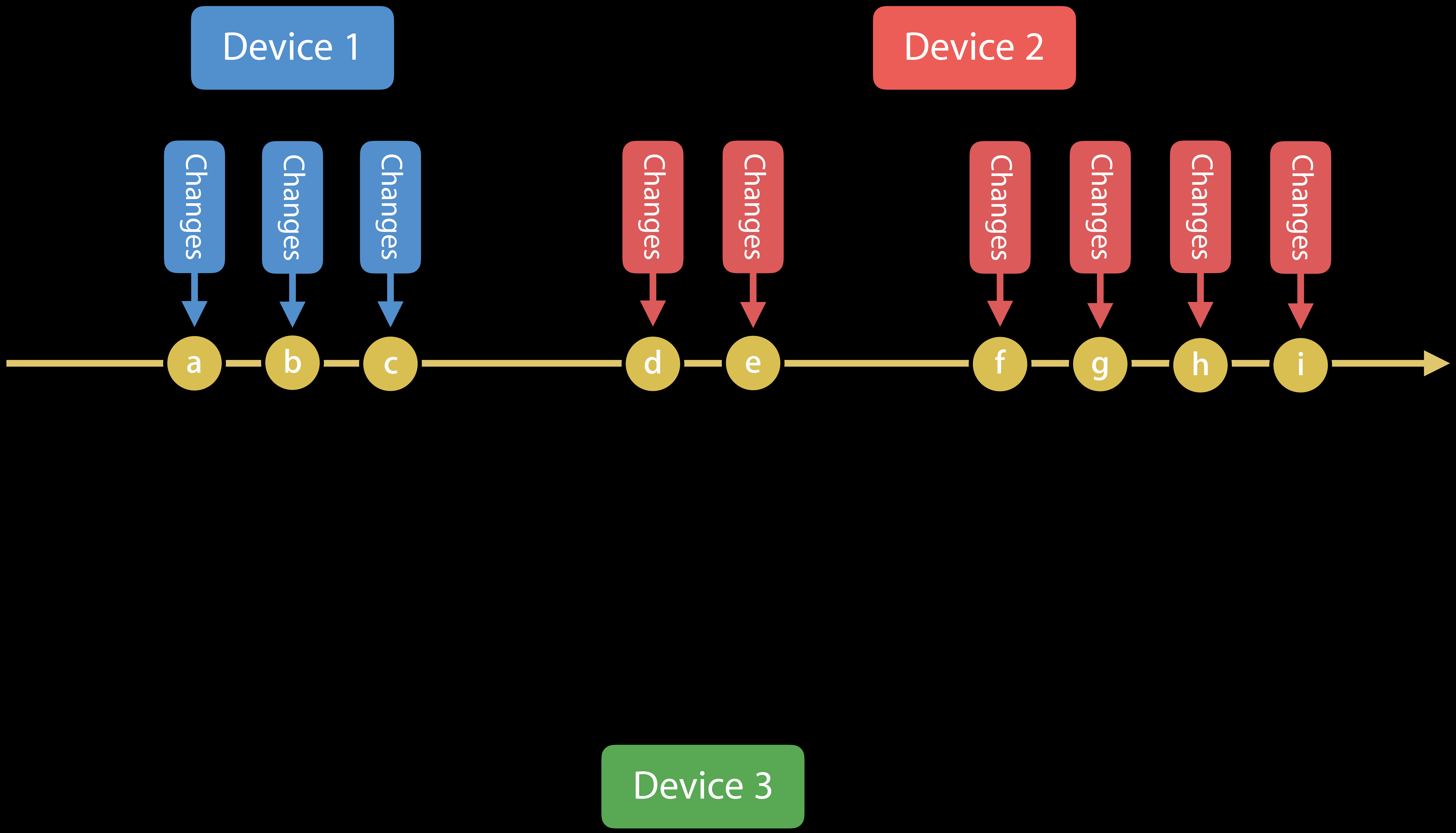
```
// Fetching database changes after a push

func fetchSharedChanges(_ callback: () -> Void) {
    let changesOperation = CKFetchDatabaseChangesOperation(
        previousServerChangeToken: sharedDBChangeToken) // previously cached

    changesOperation.fetchAllChanges = true
    changesOperation.recordZoneWithIDChangedBlock = { … } // collect zone IDs
    changesOperation.recordZoneWithIDWasDeletedBlock = { … } // delete local cache
    changesOperation.changeTokenUpdatedBlock = { … } // cache new token

    changesOperation.fetchDatabaseChangesCompletionBlock = {
        (newToken: CKServerChangeToken?, more: Bool, error: NSError?) -> Void in
        // error handling here
        self.sharedDBChangeToken = newToken // cache new token
        self.fetchZoneChanges(callback) // using CKFetchRecordZoneChangesOperation
    }
    self.sharedDB.add(changesOperation)
}
```

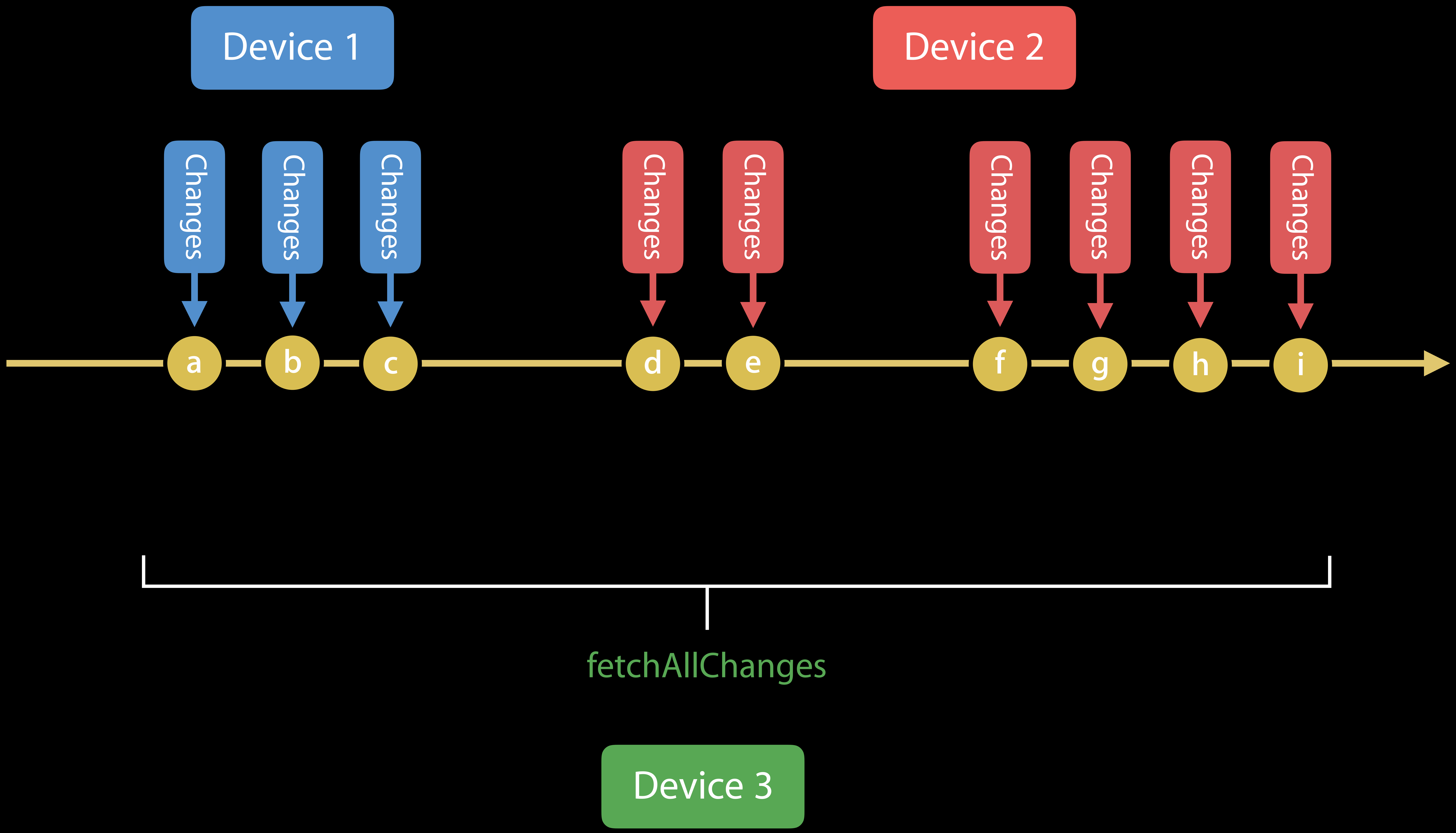
```
// Fetching database changes after a push

func fetchSharedChanges(_ callback: () -> Void) {
   let changesOperation = CKFetchDatabaseChangesOperation(
      previousServerChangeToken: sharedDBChangeToken) // previously cached

   changesOperation.fetchAllChanges = true
   changesOperation.recordZoneWithIDChangedBlock = { … } // collect zone IDs
   changesOperation.recordZoneWithIDWasDeletedBlock = { … } // delete local cache
   changesOperation.changeTokenUpdatedBlock = { … } // cache new token

   changesOperation.fetchDatabaseChangesCompletionBlock = {
      (newToken: CKServerChangeToken?, more: Bool, error: NSError?) -> Void in
      // error handling here
      self.sharedDBChangeToken = newToken // cache new token
      self.fetchZoneChanges(callback) // using CKFetchRecordZoneChangesOperation
   }
   self.sharedDB.add(changesOperation)
}
```

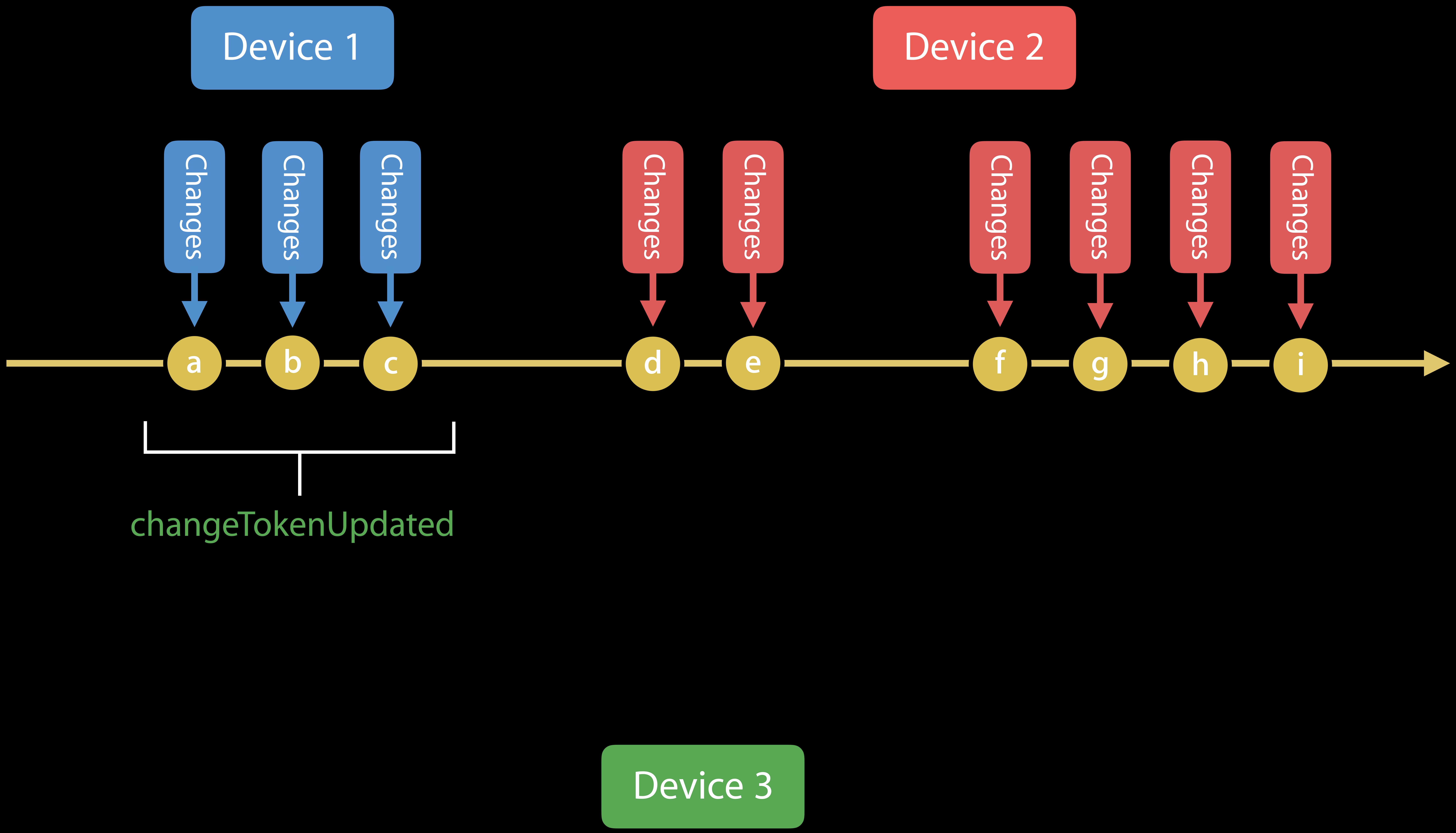
```
// Fetching database changes after a push

func fetchSharedChanges(_ callback: () -> Void) {
   let changesOperation = CKFetchDatabaseChangesOperation(
      previousServerChangeToken: sharedDBChangeToken) // previously cached

   changesOperation.fetchAllChanges = true
   changesOperation.recordZoneWithIDChangedBlock = { … } // collect zone IDs
   changesOperation.recordZoneWithIDWasDeletedBlock = { … } // delete local cache
   changesOperation.changeTokenUpdatedBlock = { … } // cache new token

   changesOperation.fetchDatabaseChangesCompletionBlock = {
      (newToken: CKServerChangeToken?, more: Bool, error: NSError?) -> Void in
      // error handling here
      self.sharedDBChangeToken = newToken // cache new token
      self.fetchZoneChanges(callback) // using CKFetchRecordZoneChangesOperation
   }
   self.sharedDB.add(changesOperation)
}
```

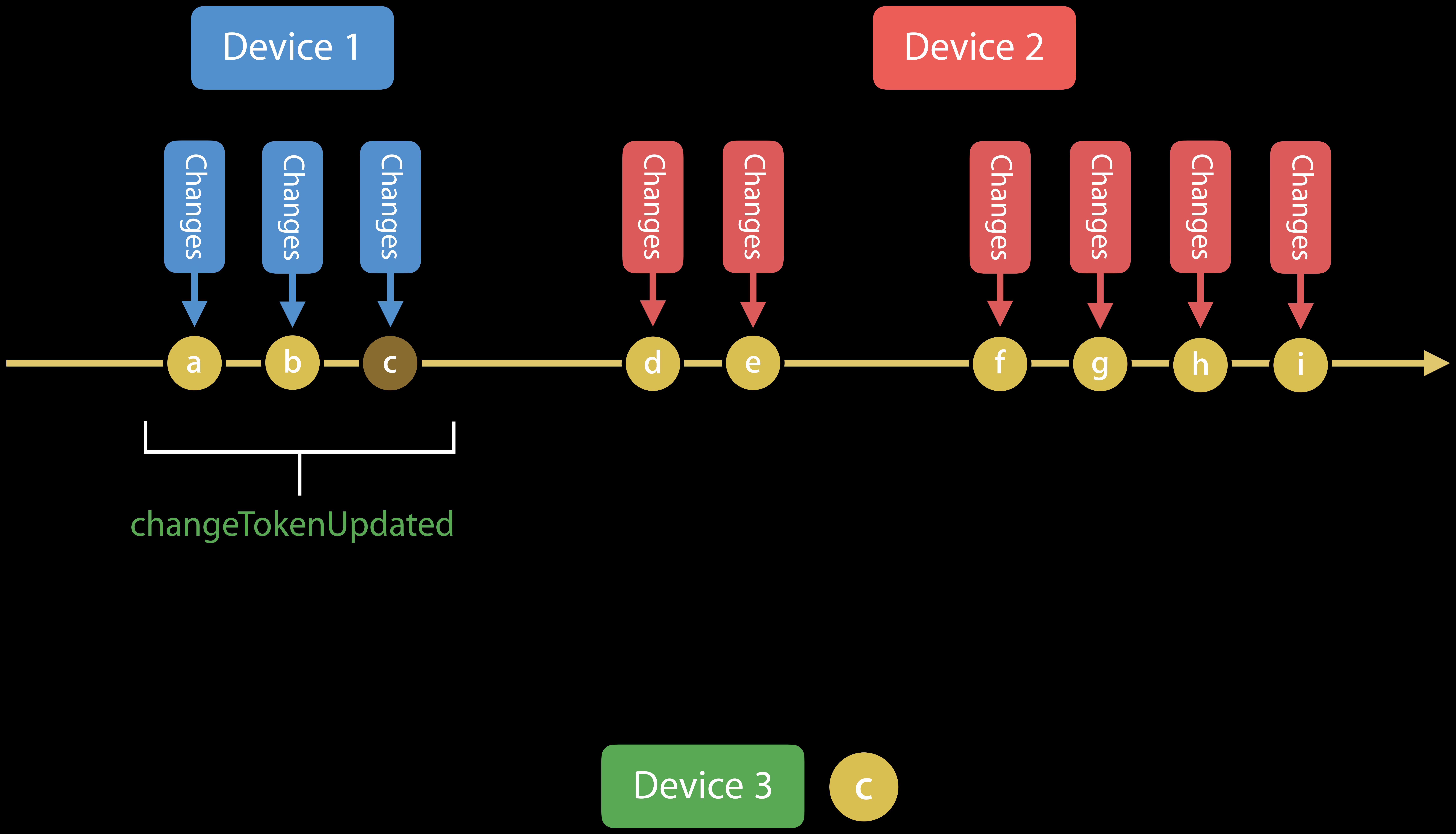

Device 1
Changes
Changes
Changes
a
b
c

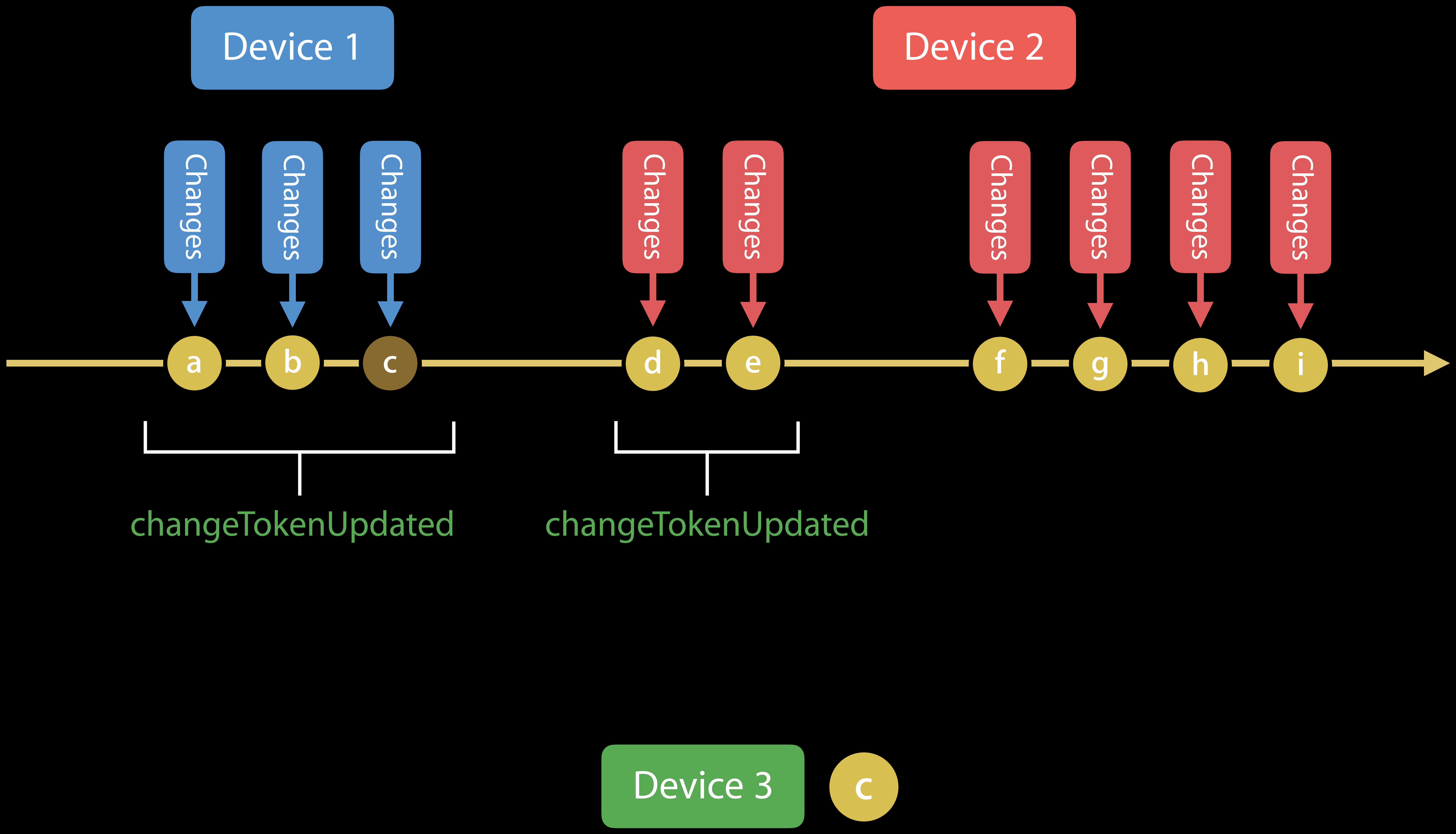

Device 1
Device 2
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
a
b
c
d
e
f
g
h
i

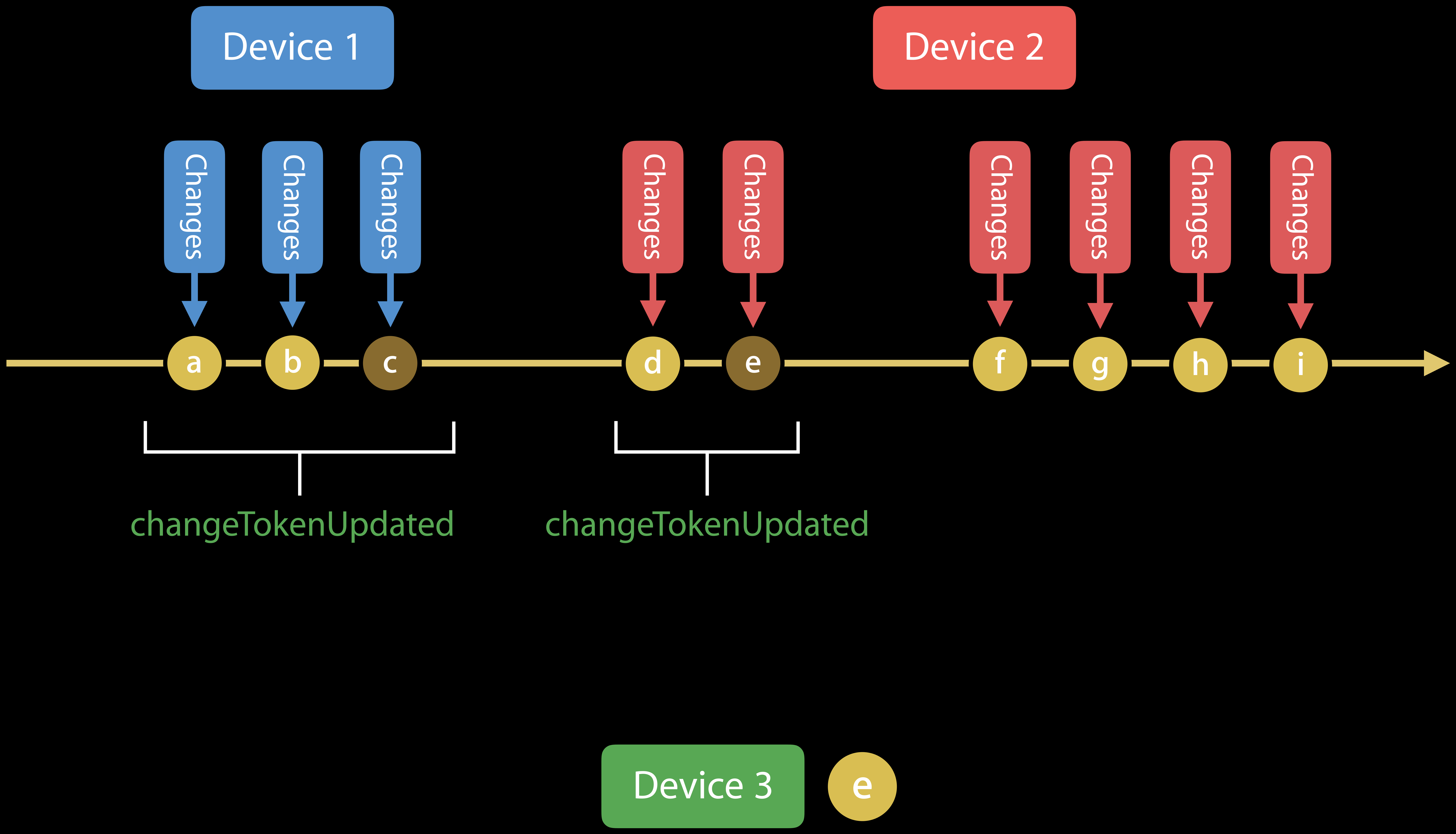

Device 1
Device 2
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
a
b
c
d
e
f
g
h
i
Device 3

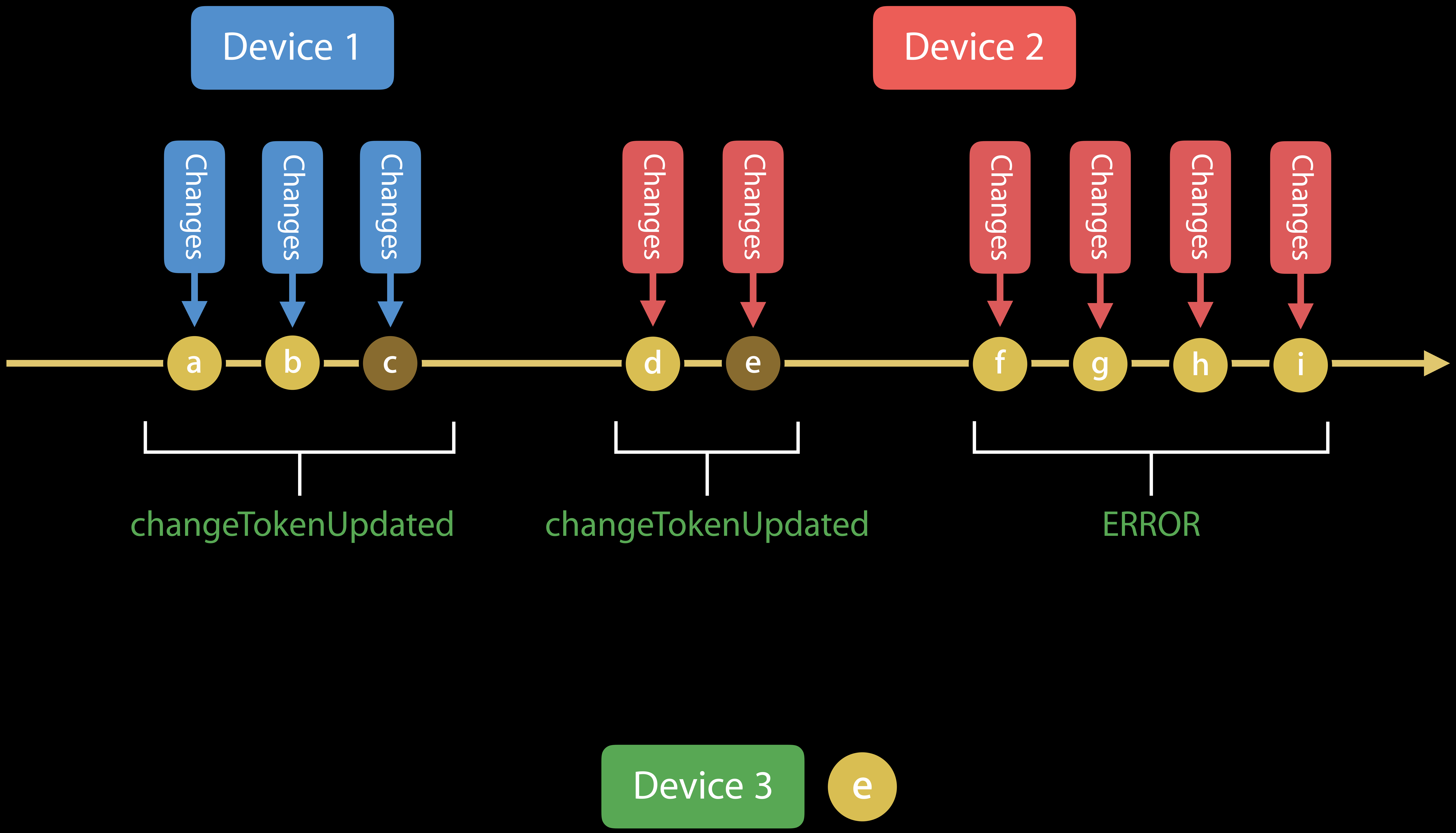

Device 1
Device 2
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
a
b
c
d
e
f
g
h
i
fetchAllChanges
Device 3

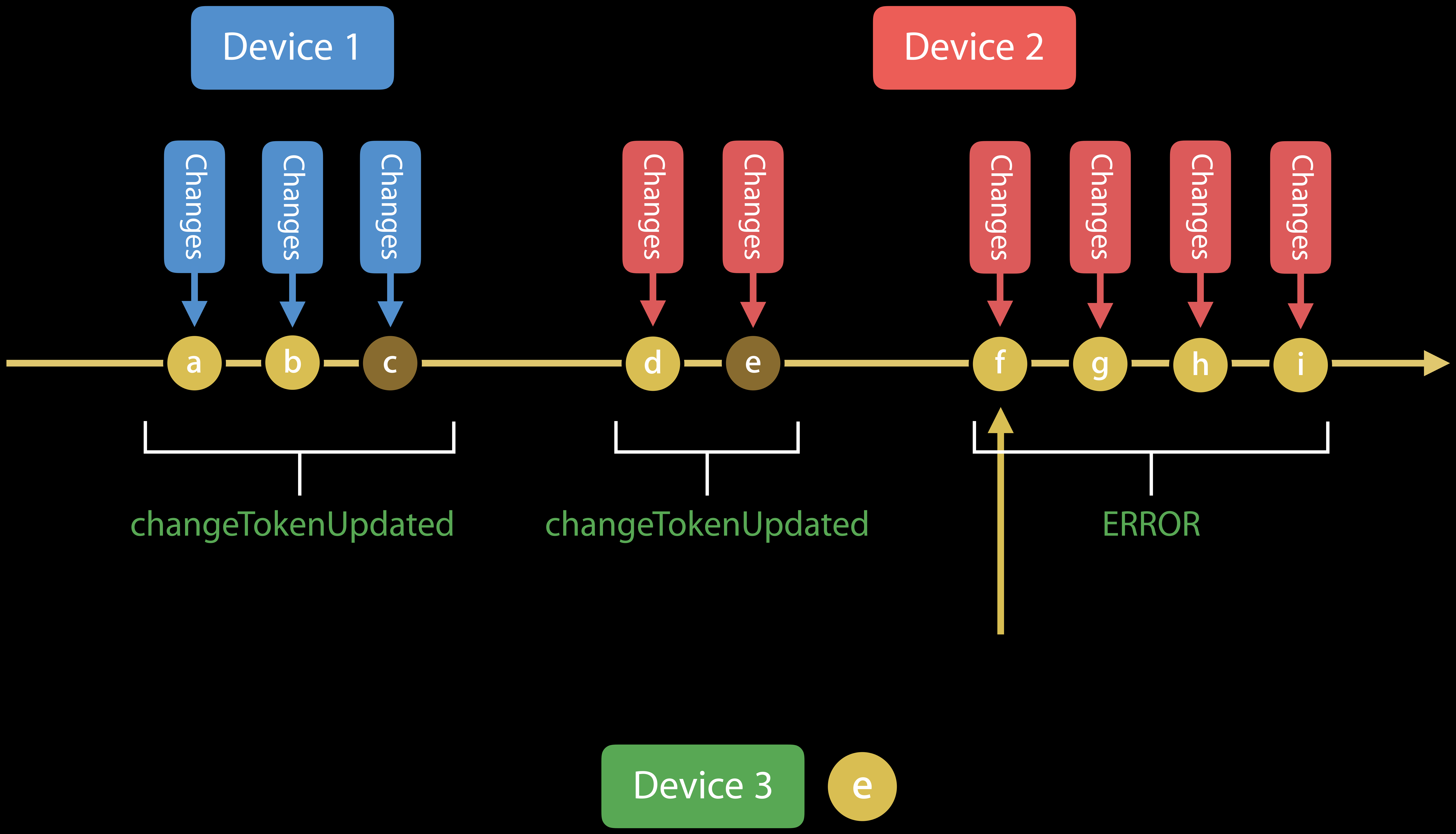

Device 1
Device 2
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
a
b
c
d
e
f
g
h
i
changeTokenUpdated
Device 3


Device 1
Device 2
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
a
b
c
d
e
f
g
h
i
changeTokenUpdated
Device 3
c


Device 1
Device 2
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
a
b
c
d
e
f
g
h
i
changeTokenUpdated
changeTokenUpdated
Device 3
c


Device 1
Device 2
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
a
b
c
d
e
f
g
h
i
changeTokenUpdated
changeTokenUpdated
Device 3
e


Device 1
Device 2
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
a
b
c
d
e
f
g
h
i
changeTokenUpdated
changeTokenUpdated
ERROR
Device 3
e


Device 1
Device 2
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
Changes
a
b
c
d
e
f
g
h
i
changeTokenUpdated
changeTokenUpdated
ERROR
Device 3
e

```
// Fetching database changes after a push

func fetchSharedChanges(_ callback: () -> Void) {
   let changesOperation = CKFetchDatabaseChangesOperation(
      previousServerChangeToken: sharedDBChangeToken) // previously cached

   changesOperation.fetchAllChanges = true
     changesOperation.recordZoneWithIDChangedBlock = { … } // collect zone IDs
   changesOperation.recordZoneWithIDWasDeletedBlock = { … } // delete local cache
   changesOperation.changeTokenUpdatedBlock = { … } // cache new token

   changesOperation.fetchDatabaseChangesCompletionBlock = {
      (newToken: CKServerChangeToken?, more: Bool, error: NSError?) -> Void in
      // error handling here
      self.sharedDBChangeToken = newToken // cache new token
      self.fetchZoneChanges(callback) // using CKFetchRecordZoneChangesOperation
   }
   self.sharedDB.add(changesOperation)
}
```

```swift
// Fetching database changes after a push

func fetchSharedChanges(_ callback: () -> Void) {
    let changesOperation = CKFetchDatabaseChangesOperation(
       previousServerChangeToken: sharedDBChangeToken) // previously cached

    changesOperation.fetchAllChanges = true
      changesOperation.recordZoneWithIDChangedBlock = { … } // collect zone IDs
    changesOperation.recordZoneWithIDWasDeletedBlock = { … } // delete local cache
    changesOperation.changeTokenUpdatedBlock = { … } // cache new token

    changesOperation.fetchDatabaseChangesCompletionBlock = {
       (newToken: CKServerChangeToken?, more: Bool, error: NSError?) -> Void in
       // error handling here
       self.sharedDBChangeToken = newToken // cache new token
       self.fetchZoneChanges(callback) // using CKFetchRecordZoneChangesOperation
    }
    self.sharedDB.add(changesOperation)
}
```

```swift
// Fetching database changes after a push

func fetchSharedChanges(_ callback: () -> Void) {
   let changesOperation = CKFetchDatabaseChangesOperation(
      previousServerChangeToken: sharedDBChangeToken) // previously cached

   changesOperation.fetchAllChanges = true
     changesOperation.recordZoneWithIDChangedBlock = { … } // collect zone IDs
   changesOperation.recordZoneWithIDWasDeletedBlock = { … } // delete local cache
   changesOperation.changeTokenUpdatedBlock = { … } // cache new token

   changesOperation.fetchDatabaseChangesCompletionBlock = {
      (newToken: CKServerChangeToken?, more: Bool, error: NSError?) -> Void in
      // error handling here
      self.sharedDBChangeToken = newToken // cache new token
      self.fetchZoneChanges(callback) // using CKFetchRecordZoneChangesOperation
   }
   self.sharedDB.add(changesOperation)
}
```

```
// Fetching database changes after a push

func fetchSharedChanges(_ callback: () -> Void) {
   let changesOperation = CKFetchDatabaseChangesOperation(
      previousServerChangeToken: sharedDBChangeToken) // previously cached

   changesOperation.fetchAllChanges = true
     changesOperation.recordZoneWithIDChangedBlock = { … } // collect zone IDs
   changesOperation.recordZoneWithIDWasDeletedBlock = { … } // delete local cache
   changesOperation.changeTokenUpdatedBlock = { … } // cache new token

   changesOperation.fetchDatabaseChangesCompletionBlock = {
      (newToken: CKServerChangeToken?, more: Bool, error: NSError?) -> Void in
      // error handling here
      self.sharedDBChangeToken = newToken // cache new token
      self.fetchZoneChanges(callback) // using CKFetchRecordZoneChangesOperation
   }
   self.sharedDB.add(changesOperation)
}
```

```swift
// Fetching database changes after a push

func fetchSharedChanges(_ callback: () -> Void) {
   let changesOperation = CKFetchDatabaseChangesOperation(
      previousServerChangeToken: sharedDBChangeToken) // previously cached

   changesOperation.fetchAllChanges = true
     changesOperation.recordZoneWithIDChangedBlock = { … } // collect zone IDs
   changesOperation.recordZoneWithIDWasDeletedBlock = { … } // delete local cache
   changesOperation.changeTokenUpdatedBlock = { … } // cache new token

   changesOperation.fetchDatabaseChangesCompletionBlock = {
      (newToken: CKServerChangeToken?, more: Bool, error: NSError?) -> Void in
      // error handling here
      self.sharedDBChangeToken = newToken // cache new token
      self.fetchZoneChanges(callback) // using CKFetchRecordZoneChangesOperation
   }
   self.sharedDB.add(changesOperation)
}
```

## Recap

# Recap

✓ Subscribing to changes

## Recap

- ✓ Subscribing to changes
- ✓ Listening for pushes

## Recap

- ✓ Subscribing to changes
- ✓ Listening for pushes
- ✓ Fetching exactly what changed

# CloudKit Best Practices

Nihar Sharma CloudKit Engineer

## Best Practices

## Best Practices

Automatic authentication

## Best Practices

Automatic authentication

CKOperation API

## Best Practices

Automatic authentication

CKOperation API

Data modeling

## Best Practices

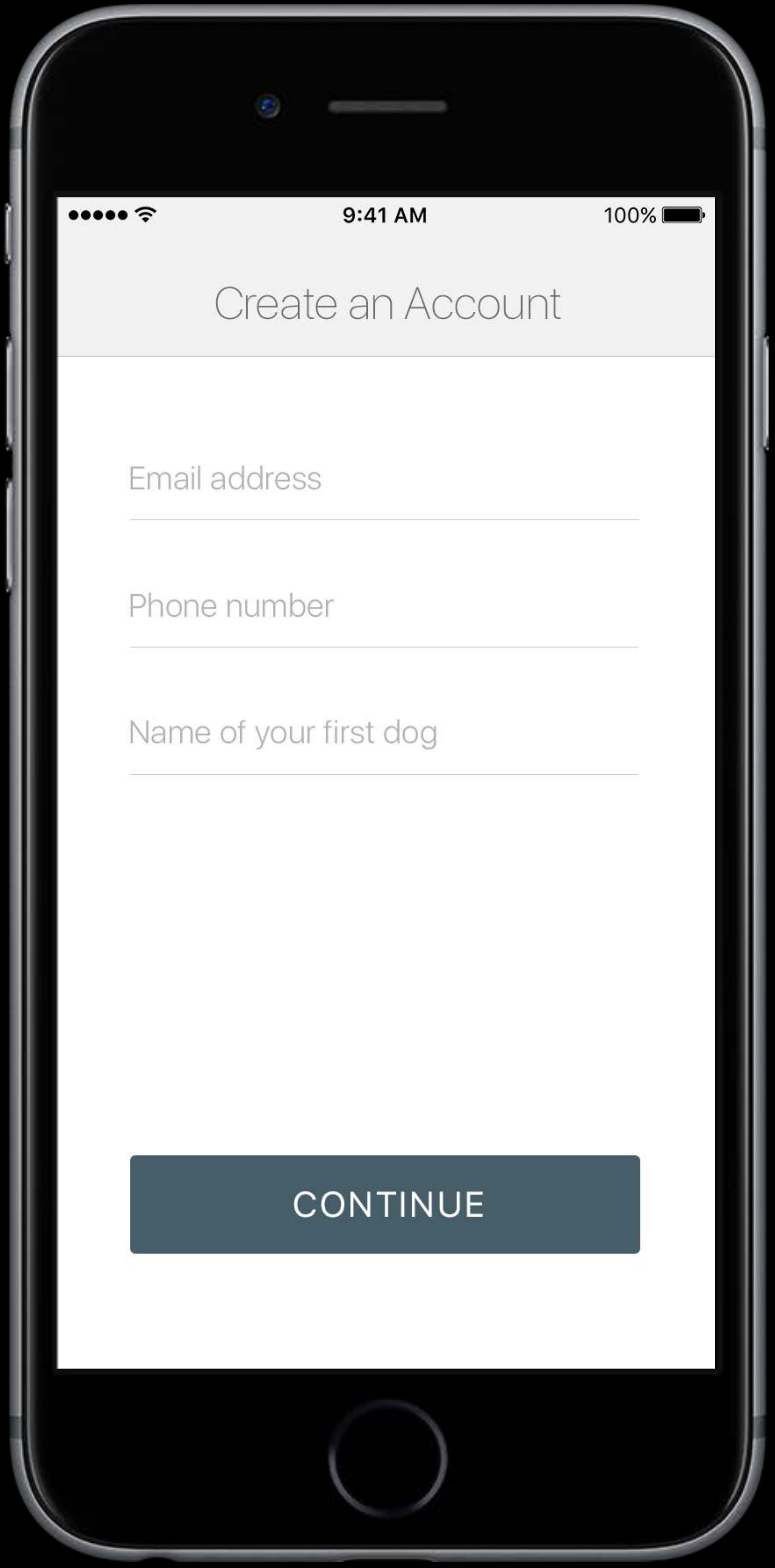
Automatic authentication

CKOperation API

Data modeling

Error handling

## Automatic Authentication


9:41 AM
100%
Create an Account
Email address
Phone number
Name of your first dog
CONTINUE

## Automatic Authentication

## Automatic Authentication

✓

CloudKit user record

## Automatic Authentication

CloudKit user record

Unique per CloudKit container

## Automatic Authentication

CloudKit user record

Unique per CloudKit container

Stable identifier

## Automatic Authentication

CloudKit user record

Unique per CloudKit container

Stable identifier

```
class CKContainer {
   public func fetchUserRecordID(completionHandler: (CKRecordID?, NSError?) -> Void)
}
```

## CKOperation

## CKOperation vs. Convenience API

## CKOperation vs. Convenience API

All convenience APIs have a `CKOperation` counterpart

## CKOperation vs. Convenience API

All convenience APIs have a `CKOperation` counterpart

Works on batches of items

# CKOperation vs. Convenience API

All convenience APIs have a `CKOperation` counterpart

Works on batches of items

```
class CKDatabase {
    public func fetch(withRecordID recordID: CKRecordID,
        completionHandler: (CKRecord?, NSError?) -> Void)

    public func save(_ record: CKRecord,
        completionHandler: (CKRecord?, NSError?) -> Void)
}
```

# CKOperation vs. Convenience API

All convenience APIs have a `CKOperation` counterpart

Works on batches of items

```
class CKDatabase {
   public func fetch(withRecordID recordID: CKRecordID,
      completionHandler: (CKRecord?, NSError?) -> Void)

   public func save(_ record: CKRecord,
      completionHandler: (CKRecord?, NSError?) -> Void)
}
```

```
CKFetchRecordsOperation
CKModifyRecordsOperation
```

## CKOperation

## CKOperation

```
class CKOperation : Operation
```

# CKOperation

```
class CKOperation : Operation
```

- Set up dependencies

# CKOperation

```
class CKOperation : Operation
```

- Set up dependencies
- Quality of service and queue priorities

# CKOperation

```
class CKOperation : Operation
```

- Set up dependencies
- Quality of service and queue priorities
- Cancellation

# CKOperation

```
class CKOperation : Operation
```

- Set up dependencies
- Quality of service and queue priorities
- Cancellation

## CKOperation

## CKOperation

```
class CKOperation : Operation
```

## CKOperation

```
class CKOperation : Operation
```

- Configurability

# CKOperation

```
class CKOperation : Operation
```

- Configurability
- Resource optimization

## CKOperation

```
class CKOperation : Operation
```

- Configurability
- Resource optimization
- Lifetime management

# CKOperation

Configurability

# CKOperation

## Configurability

Cellular access

## CKOperation

### Configurability

Cellular access

Fetch partial records

## CKOperation

### Configurability

Cellular access

Fetch partial records

Limit number of results

## CKOperation

### Configurability

Cellular access

Fetch partial records

Limit number of results

Progress reporting

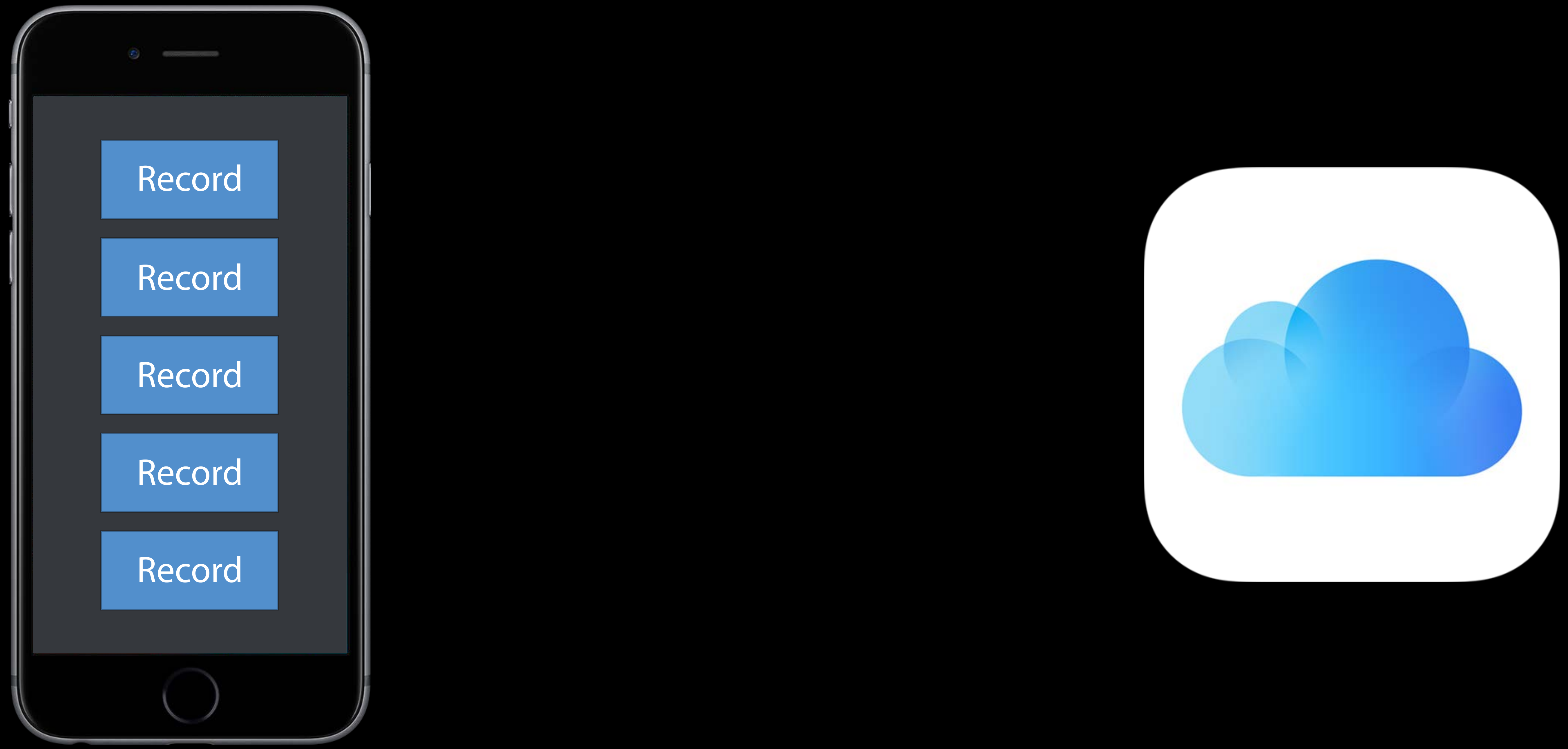
## CKOperation

Resource optimization

# CKOperation

## Resource optimization

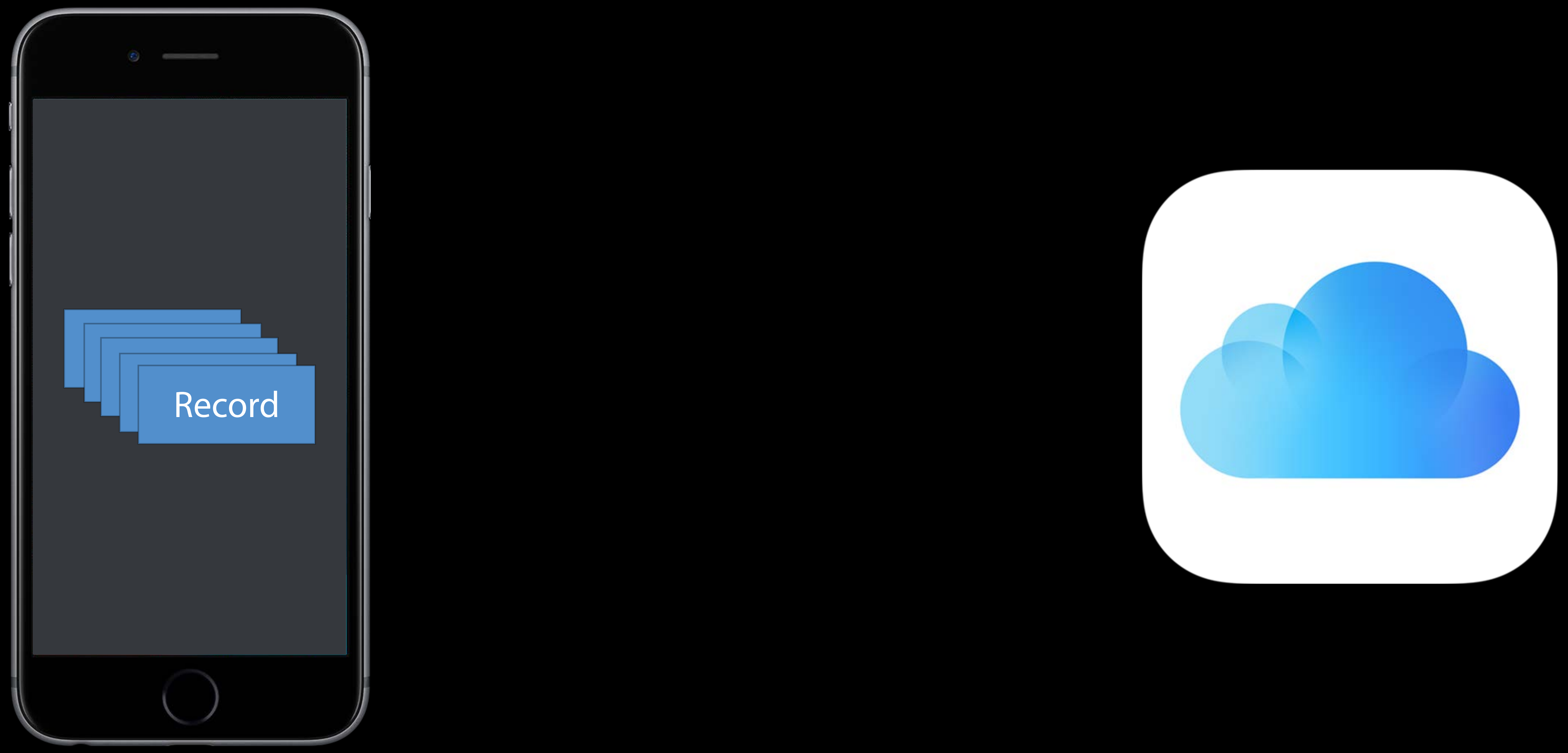
Minimize network requests



# CKOperation

## Resource optimization

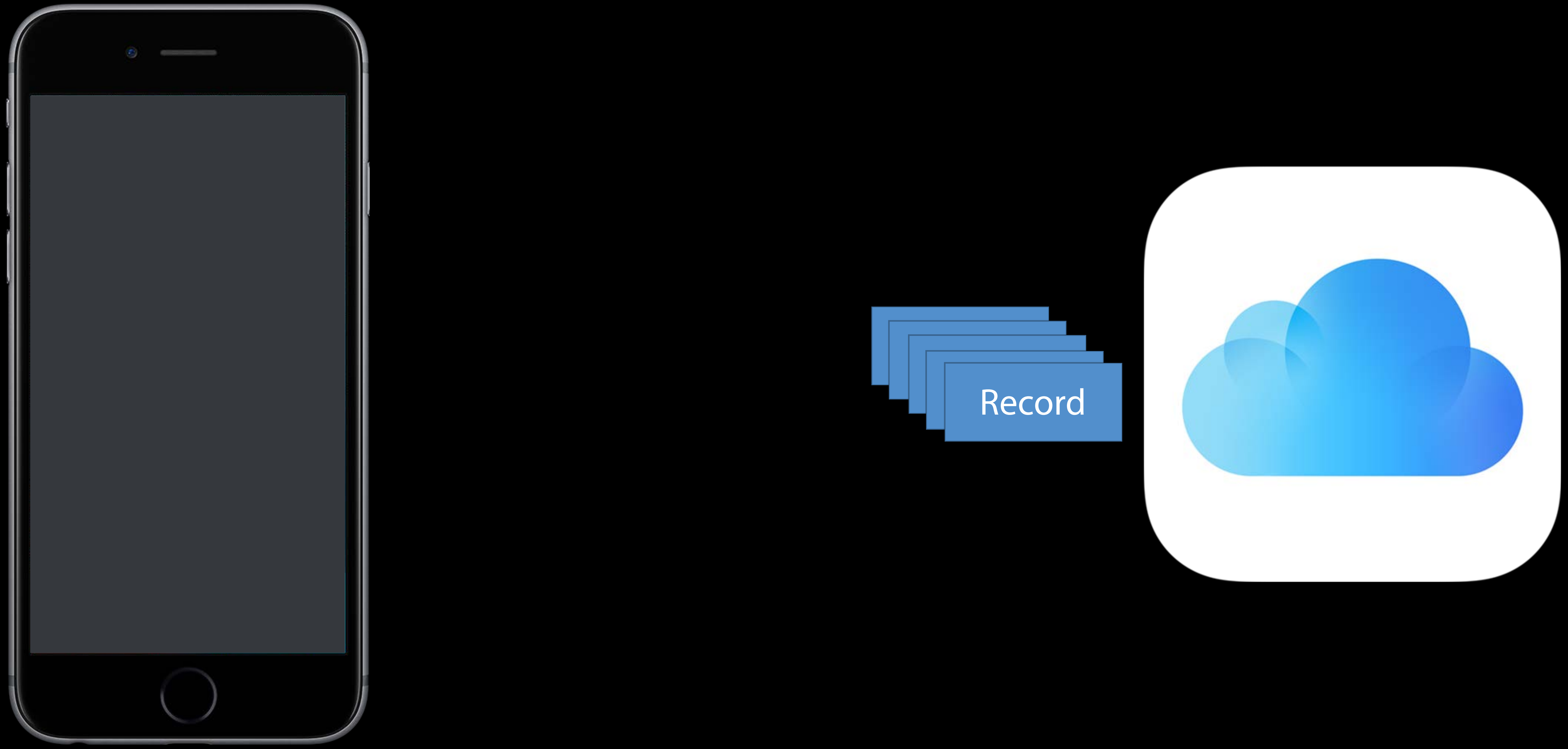
Minimize network requests



## CKOperation

### Resource optimization

Minimize network requests

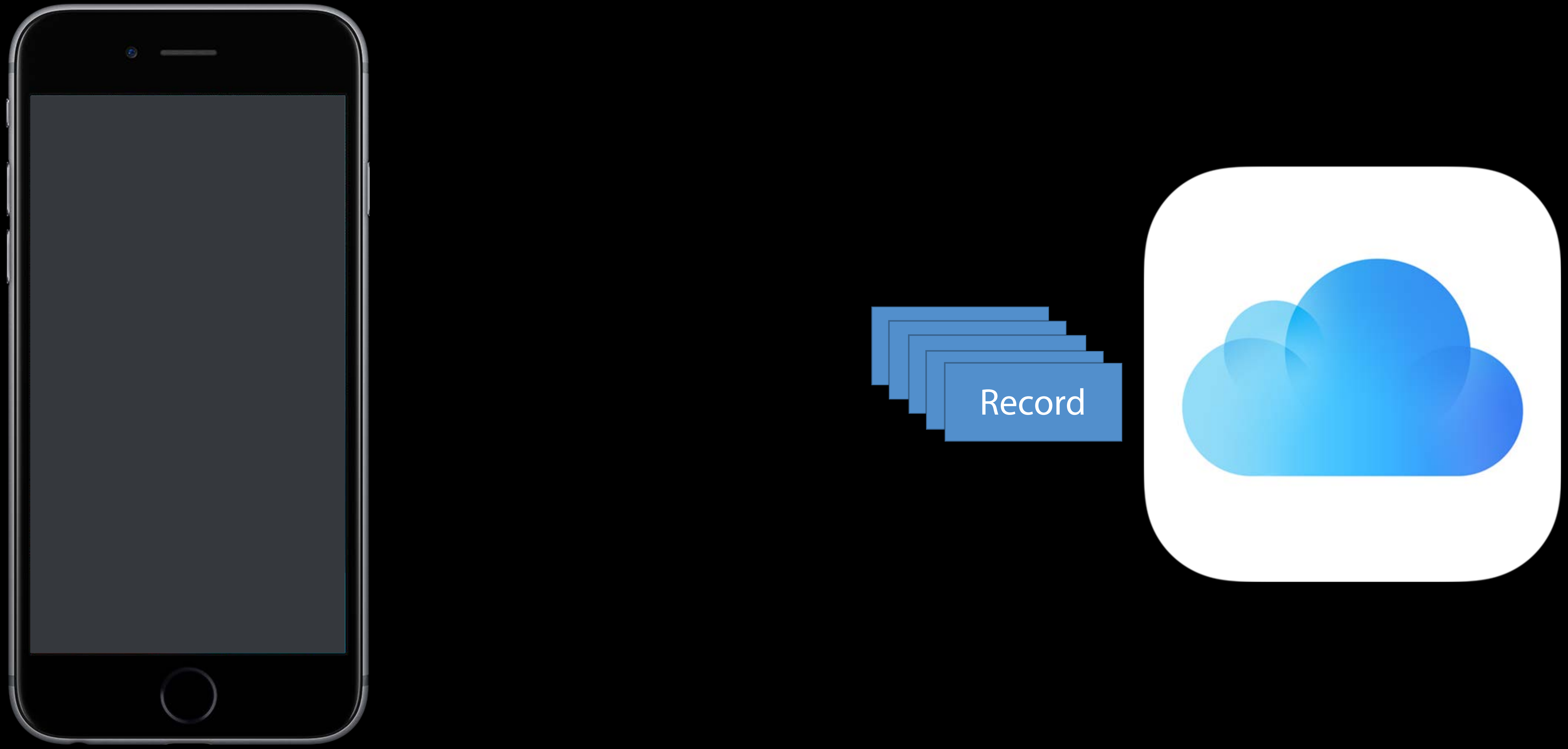


# CKOperation

## Resource optimization

Minimize network requests

System resources



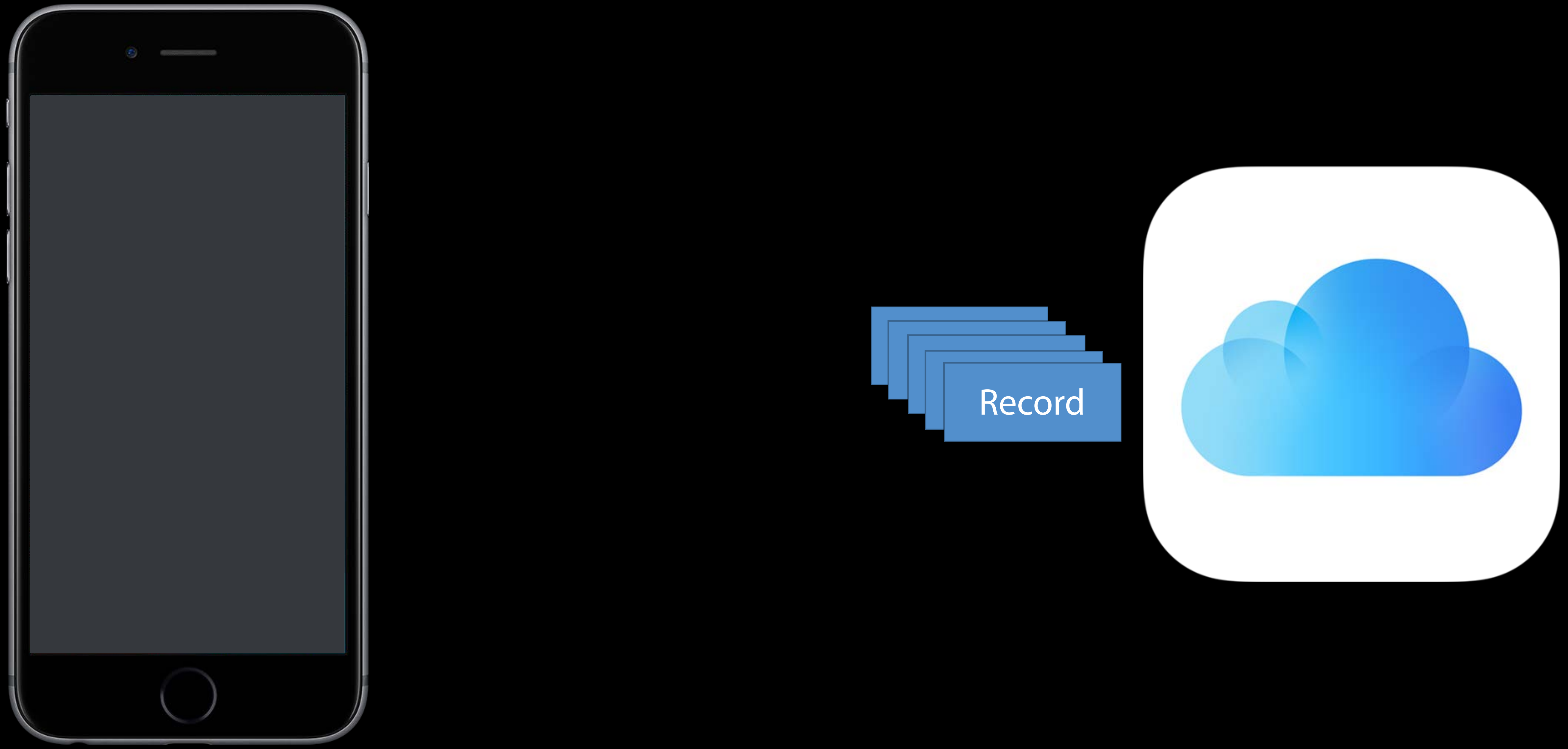
# CKOperation

## Resource optimization

Minimize network requests

System resources

Request quota



## CKOperation

Resource optimization

# CKOperation

## Resource optimization

Discretionary behavior opt-in

# CKOperation

## Resource optimization

Discretionary behavior opt-in

```
public enum QualityOfService : Int {
    case userInteractive
    case userInitiated
    case utility
    case background
    case `default`
}
```

# CKOperation

## Resource optimization

✓

Discretionary behavior opt-in

```
public enum QualityOfService : Int {
    case userInteractive
    case userInitiated
    case utility
    case background
    case `default`
}
```

# CKOperation

## Quality of Service

```
public enum QualityOfService : Int {
    case userInteractive
    case userInitiated
    case utility
    case background
    case `default`
}
```

# CKOperation

## Quality of Service

Network failures are retried

```
public enum QualityOfService : Int {
    case userInteractive
    case userInitiated
    case utility
    case background
    case `default`
}
```

# CKOperation

## Quality of Service

Network failures are retried

7-day resource timeout

```
public enum QualityOfService : Int {
    case userInteractive
    case userInitiated
    case utility
    case background
    case `default`
}
```

## CKOperation

Lifetime management

# CKOperation

## Lifetime management

App suspension or force quit

## CKOperation

### Lifetime management

App suspension or force quit

User-initiated or long-running updates

## CKOperation

### Lifetime management

App suspension or force quit

User-initiated or long-running updates

Long-lived operations

## CKOperation

Lifetime management

NEW

App suspension or force quit

User-initiated or long-running updates

Long-lived operations

## CKOperation

Lifetime management

NEW

App suspension or force quit

User-initiated or long-running updates

Long-lived operations

```
public class CKOperation : Operation {
    public var operationID: String { get }
    public var isLongLived: Bool
}
```

## Long-Lived Operations

How to use them

## Long-Lived Operations

### How to use them

Issuing Long-Lived Operations

- Initialize a CKOperation
- Set the isLongLived flag
- Set callbacks on the operation
- Run the operation

## Long-Lived Operations

### How to use them

Issuing Long-Lived Operations

- Initialize a CKOperation
- Set the isLongLived flag
- Set callbacks on the operation
- Run the operation

Resuming Long-Lived Operations

- Fetch the long-lived operation from CKContainer
- Set callbacks on operation
- Run the operation

## Long-Lived Operations

### How to use them

NEW ish

Issuing Long-Lived Operations

- Initialize a CKOperation
- Set the isLongLived flag
- Set callbacks on the operation
- Run the operation

Resuming Long-Lived Operations

- Fetch the long-lived operation from CKContainer
- Set callbacks on operation
- Run the operation

## Long-Lived Operations

### Create and run the operation

NEW ish

```
let myOp = CKFetchRecordsOperation(recordIDs: [myRecordID])

myOp.isLongLived = true
let myOpID = myOp.operationID

// Set callbacks
myOp.fetchRecordsCompletionBlock = {(records, error) in ... }

// Resume the operation
CKContainer.default().privateCloudDatabase.add(myOp)
```

## Long-Lived Operations

### Create and run the operation

NEW ish

```
let myOp = CKFetchRecordsOperation(recordIDs: [myRecordID])

myOp.isLongLived = true
let myOpID = myOp.operationID

// Set callbacks
myOp.fetchRecordsCompletionBlock = {(records, error) in ... }

// Resume the operation
CKContainer.default().privateCloudDatabase.add(myOp)
```

# Long-Lived Operations

## Resume the operation

NEW ish

```
CKContainer.default().fetchLongLivedOperation(withID: myOpID) {
    (longLivedOp: CKOperation?, error: NSError?) in

    let myFetchRecordsOp = longLivedOp as! CKFetchRecordsOperation

    // Set callbacks
    myFetchRecordsOp.fetchRecordsCompletionBlock = {(records, error) in ... }

    // Resume the operation
    CKContainer.default().privateCloudDatabase.add(myFetchRecordsOp)
}
```

# Long-Lived Operations

## Resume the operation

NEW ish

```
CKContainer.default().fetchLongLivedOperation(withID: myOpID) {
    (longLivedOp: CKOperation?, error: NSError?) in

    let myFetchRecordsOp = longLivedOp as! CKFetchRecordsOperation

    // Set callbacks
    myFetchRecordsOp.fetchRecordsCompletionBlock = {(records, error) in ... }

    // Resume the operation
    CKContainer.default().privateCloudDatabase.add(myFetchRecordsOp)
}
```

## Long-Lived Operations

Resume the operation

NEW ish

```
CKContainer.default().fetchLongLivedOperation(withID: myOpID) {
    (longLivedOp: CKOperation?, error: NSError?) in

    let myFetchRecordsOp = longLivedOp as! CKFetchRecordsOperation

    // Set callbacks
    myFetchRecordsOp.fetchRecordsCompletionBlock = {(records, error) in ... }

    // Resume the operation
    CKContainer.default().privateCloudDatabase.add(myFetchRecordsOp)
}
```

## Long-Lived Operations

### Notes

NEW ish

Operations are cleaned up

## Long-Lived Operations

### Notes

NEW ish

Operations are cleaned up

- Once their completion block is called

## Long-Lived Operations

### Notes

NEW ish

Operations are cleaned up

- Once their completion block is called
- 24 hours after they complete

## Data Modeling

## Data Modeling

# Data Modeling

Schema redundancies

## Data Modeling

Schema redundancies

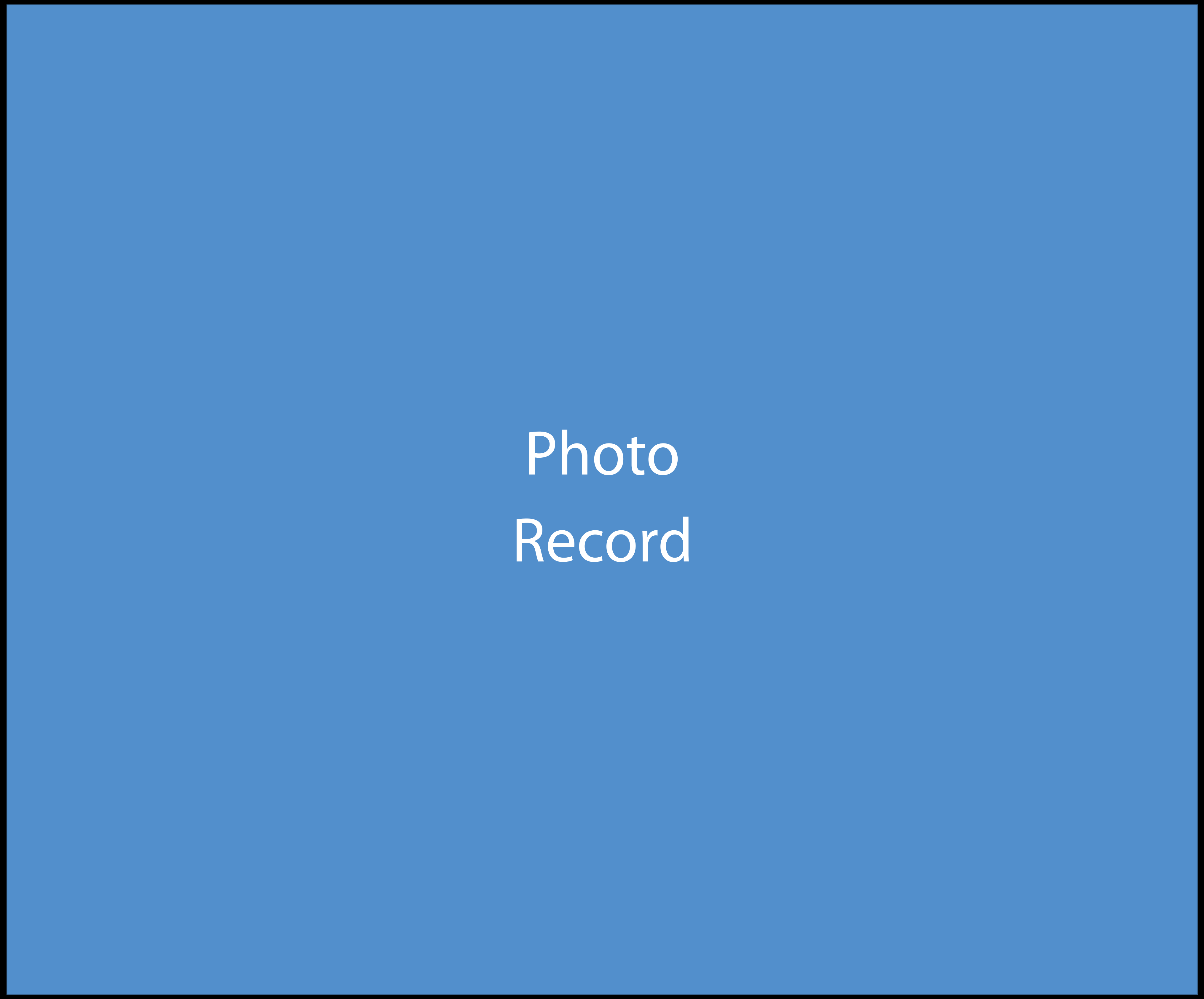
CKReferences

## Data Modeling

- Schema redundancies
- CKReferences
- Parent references

## Schema Redundancies

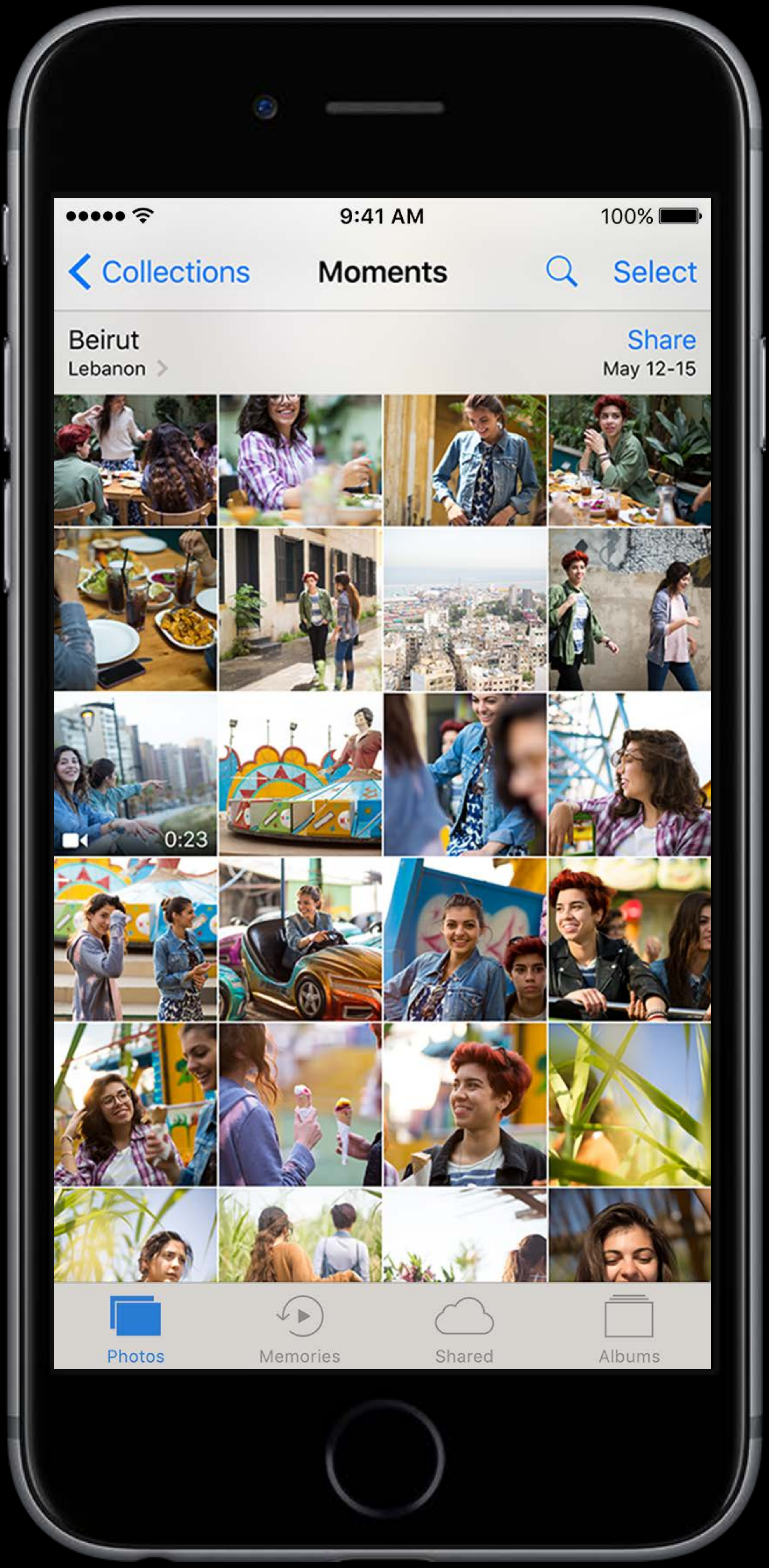
## Schema Redundancies



## Schema Redundancies

## Schema Redundancies

“Photo” =



9:41 AM
100%
Collections
Moments
Select
Beirut
Lebanon
Share
May 12-15
0:23
Photos
Memories
Shared
Albums

# Schema Redundancies

“Photo” =

## Schema Redundancies

“Photo” =

## Schema Redundancies

“Photo” =

“thumbnail1024” =

## Schema Redundancies

Optimized download

## Schema Redundancies

Optimized download

```
let myOp = CKQueryOperation(query: myQuery)
myOp.desiredKeys = ["thumbnail1024"]
myOp.resultsLimit = 10
myOp.sortDescriptors = […]
…
CKContainer.default().privateCloudDatabase.add(myOp)
```

## Schema Redundancies

Optimized download

## Schema Redundancies

Optimized download

✓ Fetch only what’s needed

## Schema Redundancies

### Optimized download

- ✓ Fetch only what’s needed
- ✓ CKFetchRecordZoneChangesOperation, CKFetchRecordsOperation

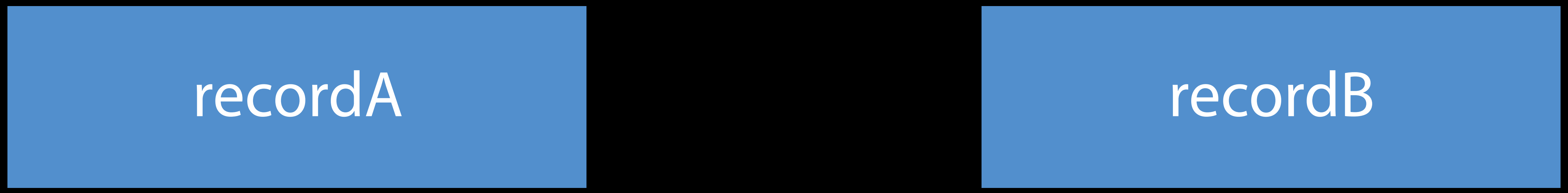
## Schema Redundancies

Optimized download

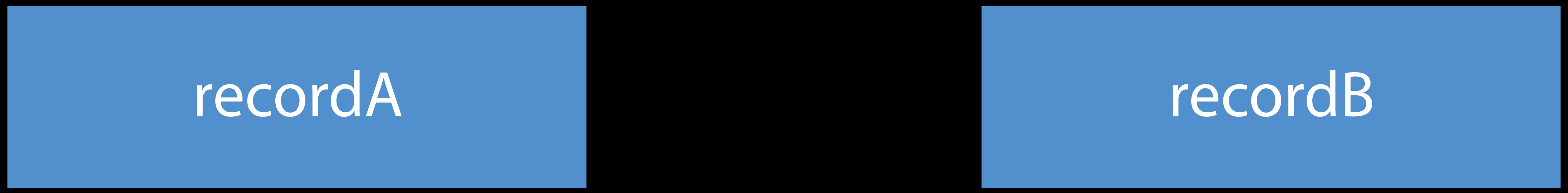
- ✓ Fetch only what's needed
- ✓ CKFetchRecordZoneChangesOperation, CKFetchRecordsOperation
- ✓ Dynamic UI

## CKReferences

## CKReferences

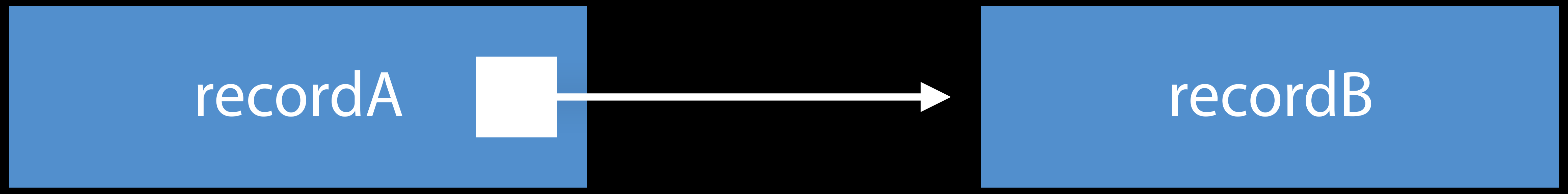


## CKReferences



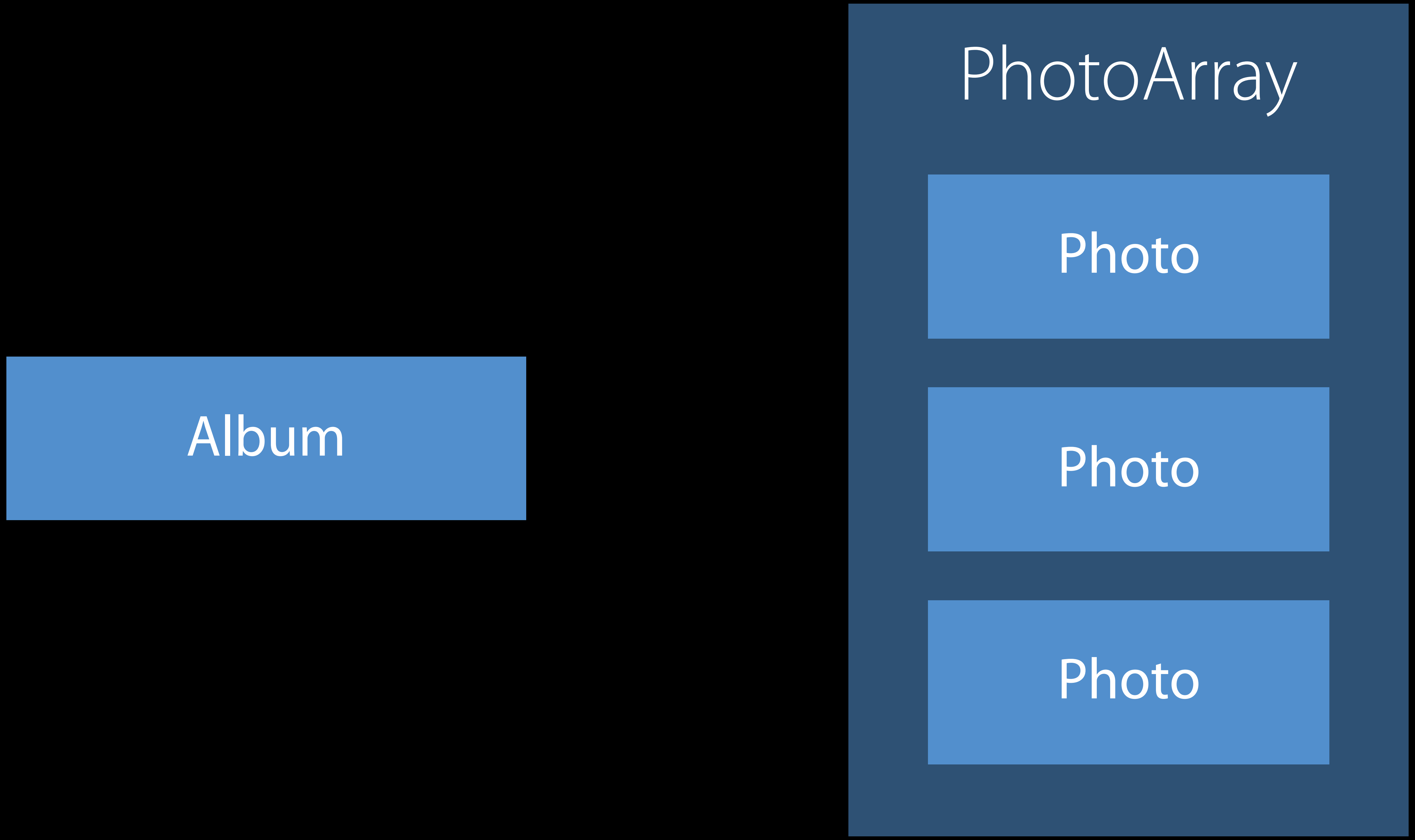
```
recordA["MyReference"] = CKReference(record: recordB, action: .None)
```

## CKReferences

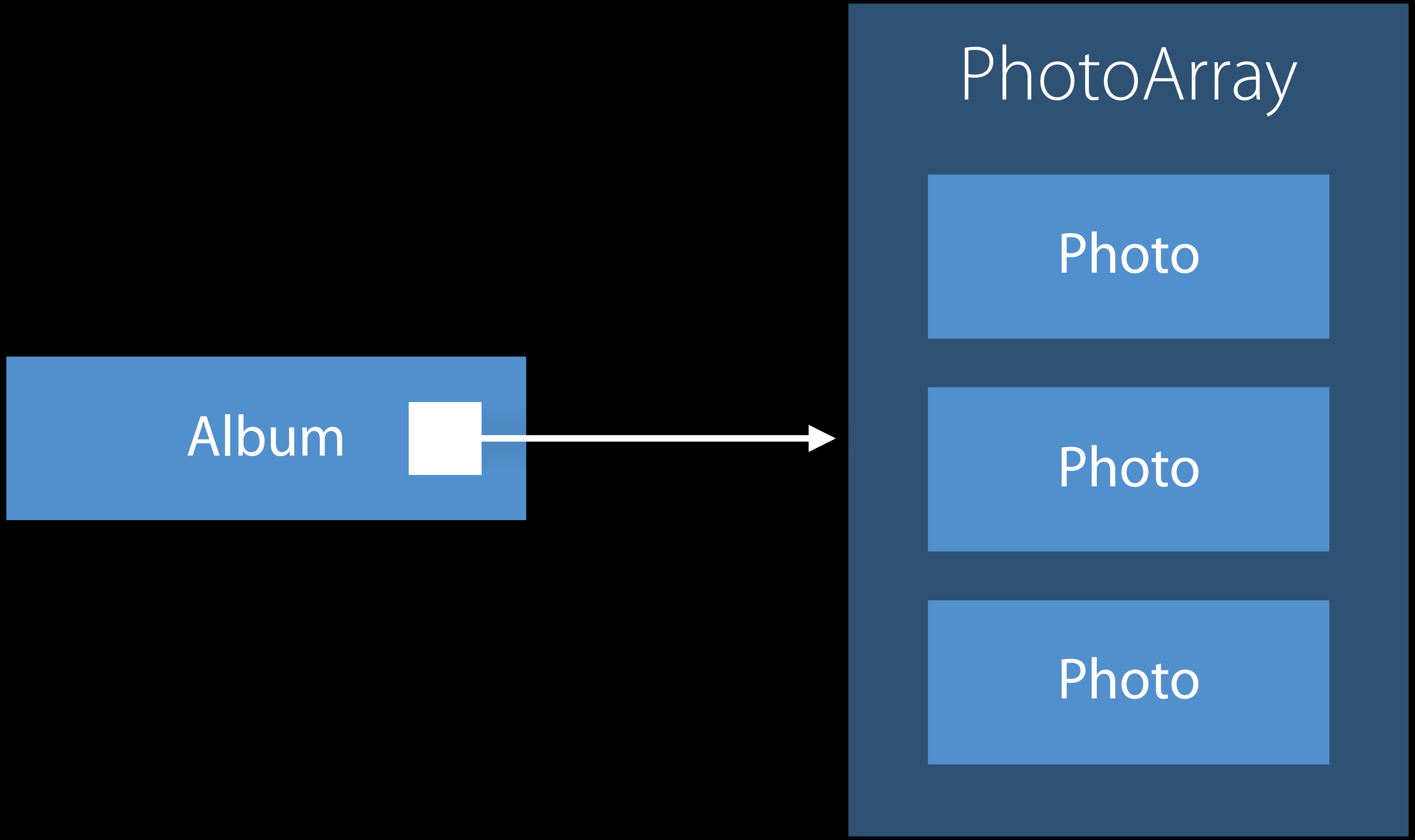


```
recordA["MyReference"] = CKReference(record: recordB, action: .None)
```

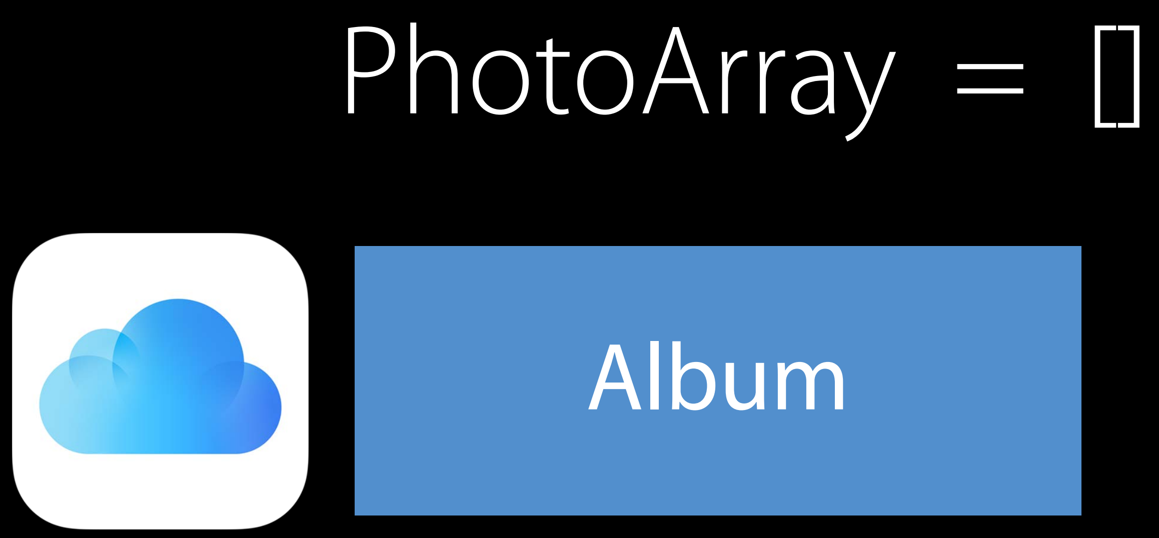
# CKReferences



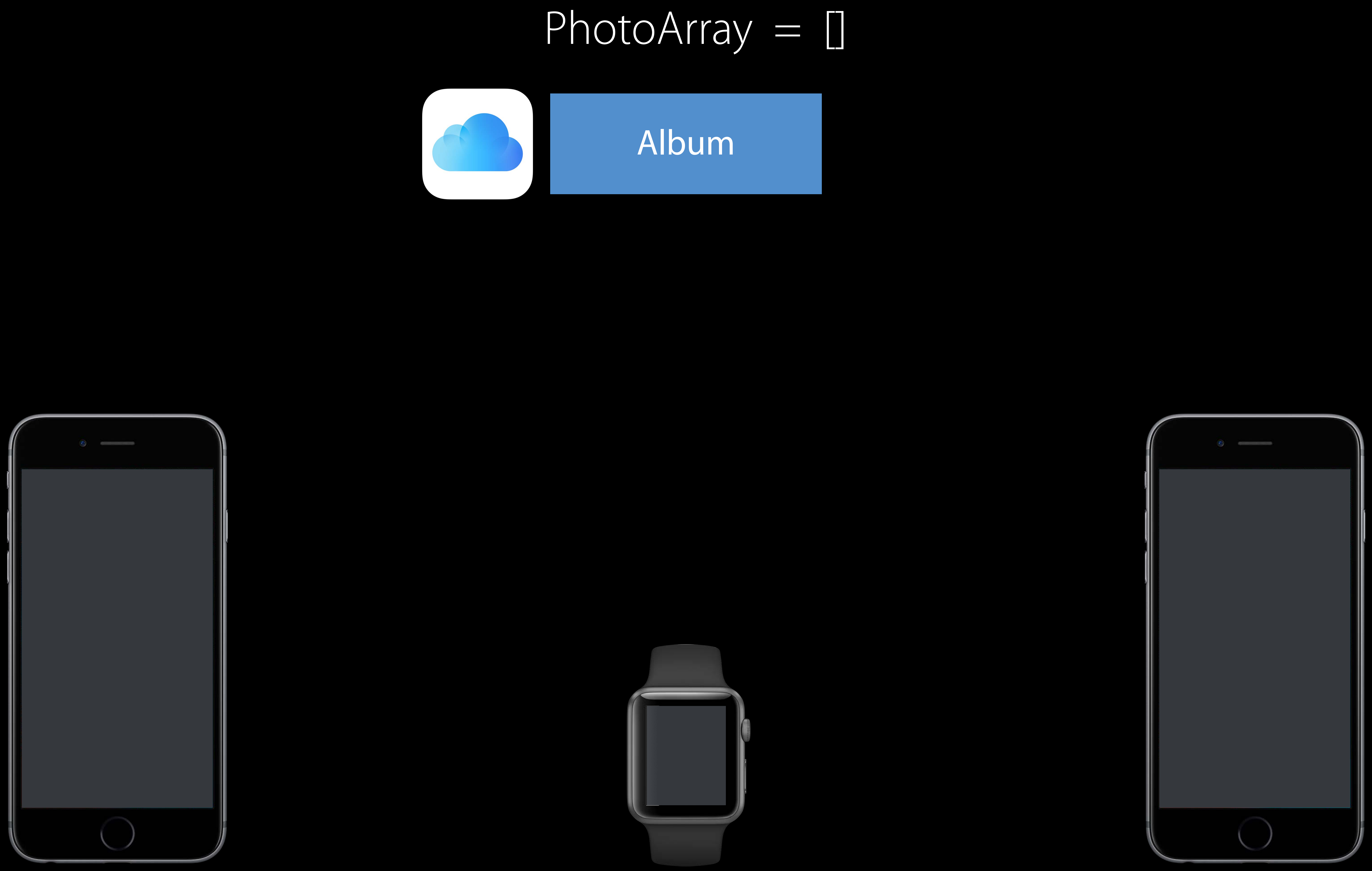
# CKReferences



# CKReferences

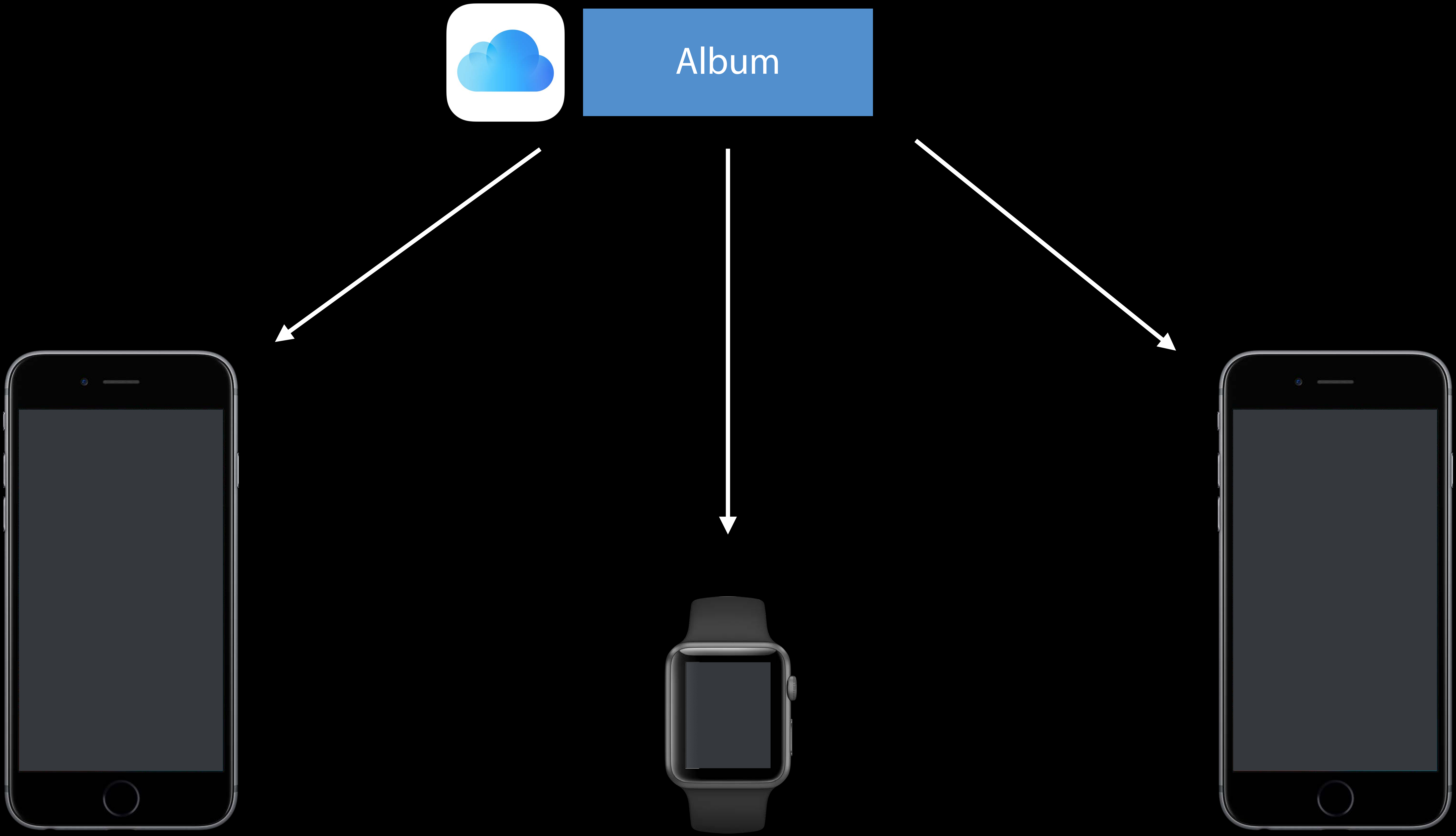


# CKReferences



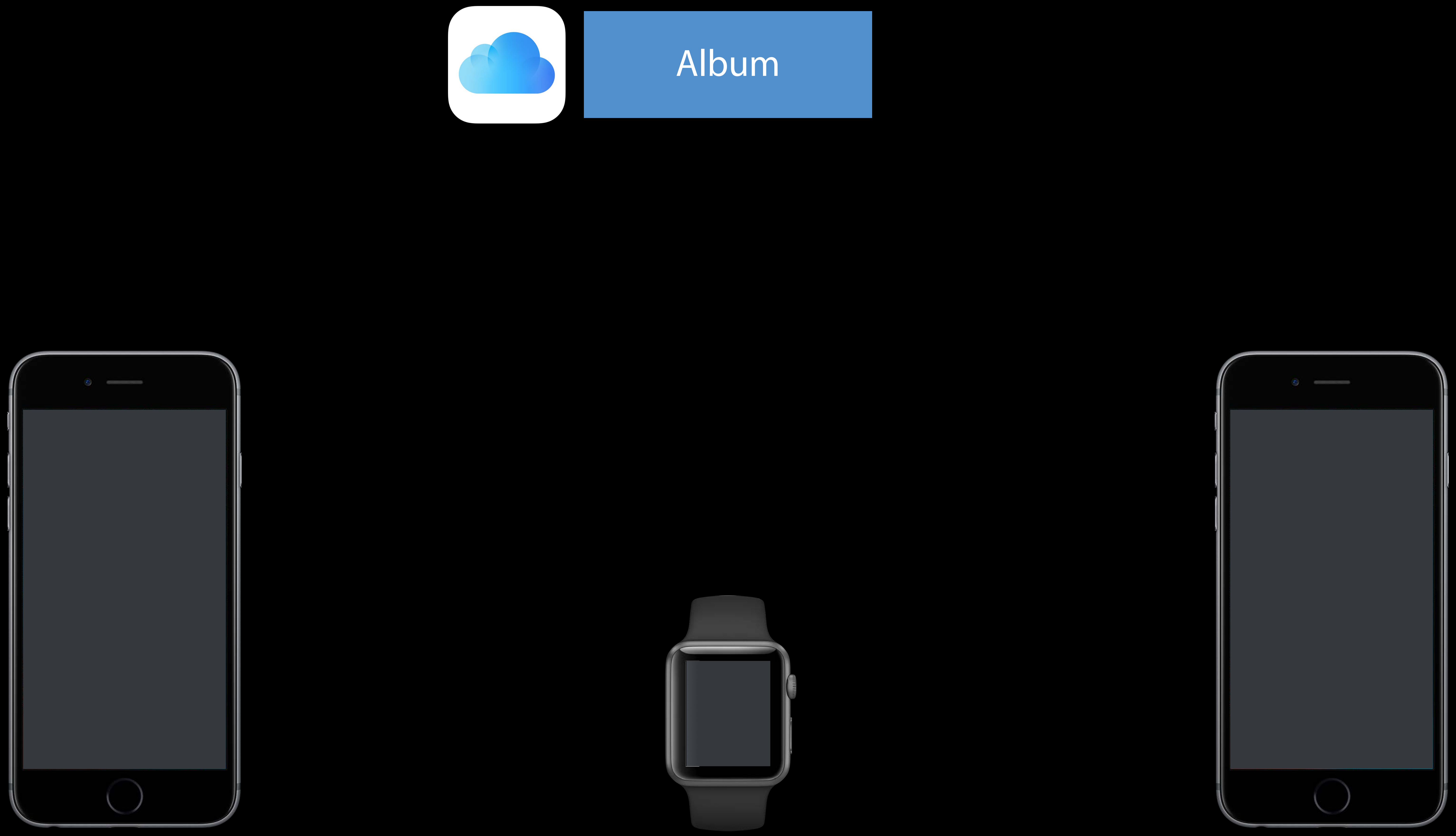
# CKReferences

PhotoArray = []

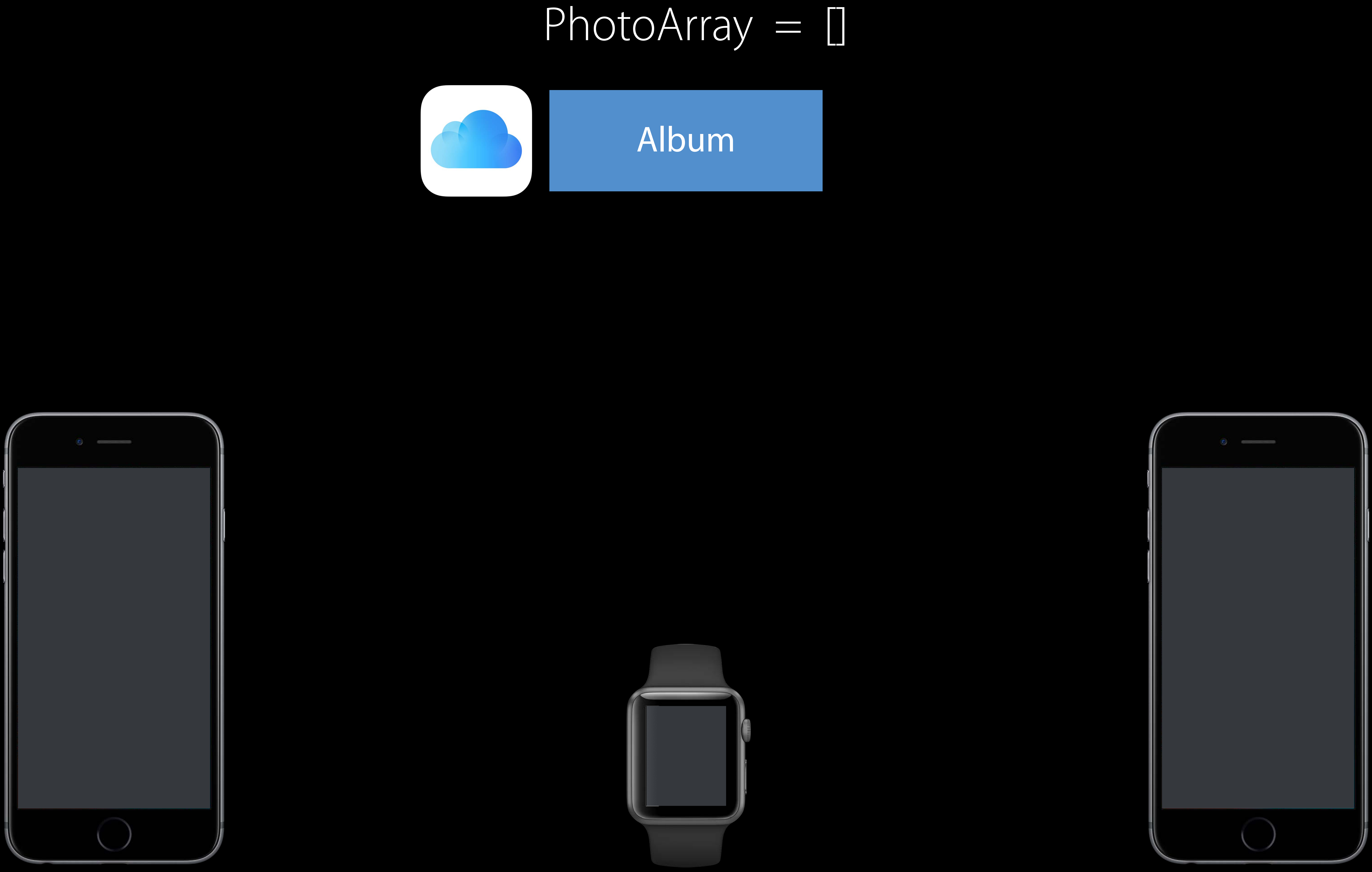


# CKReferences

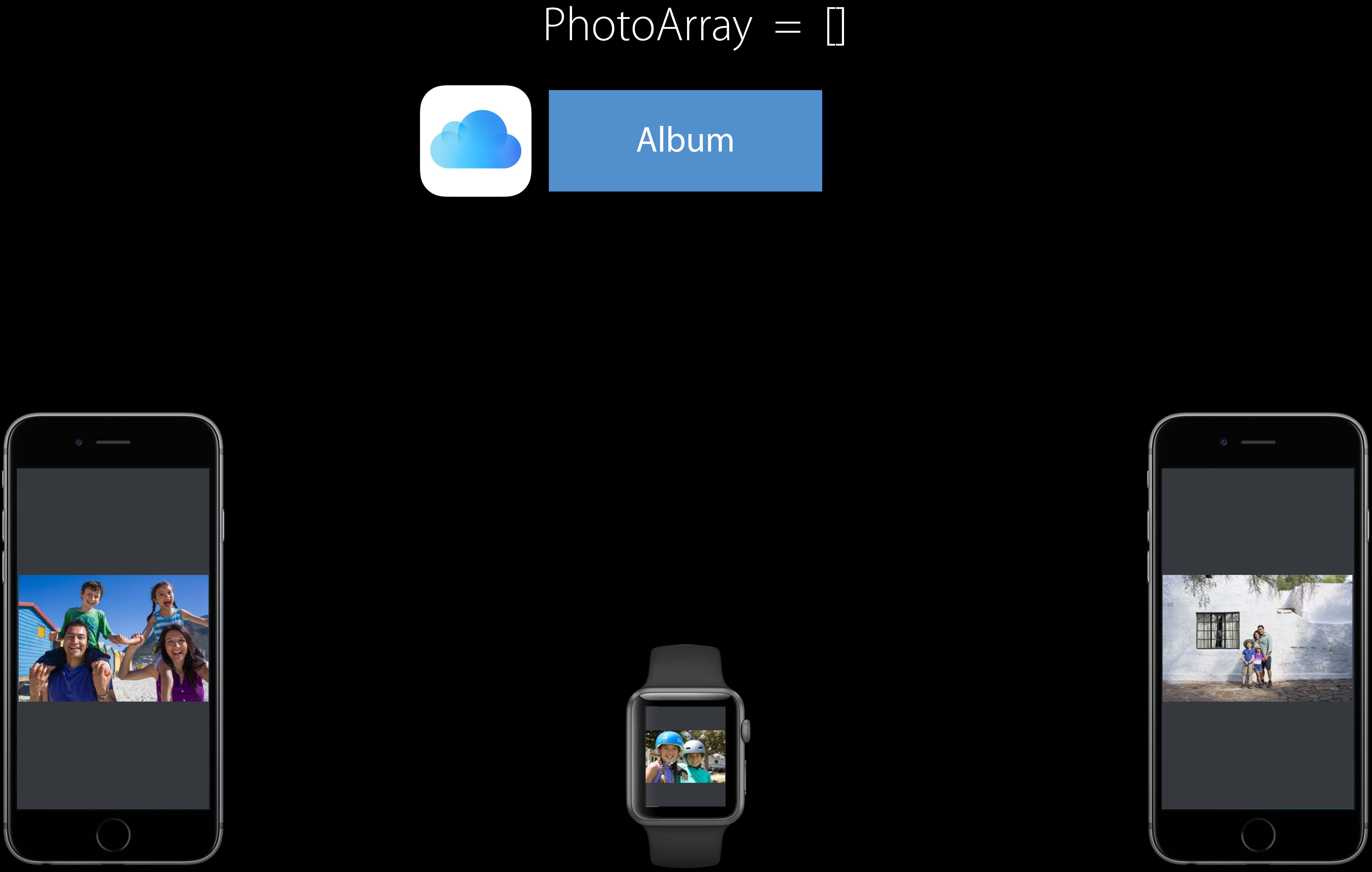
PhotoArray = []



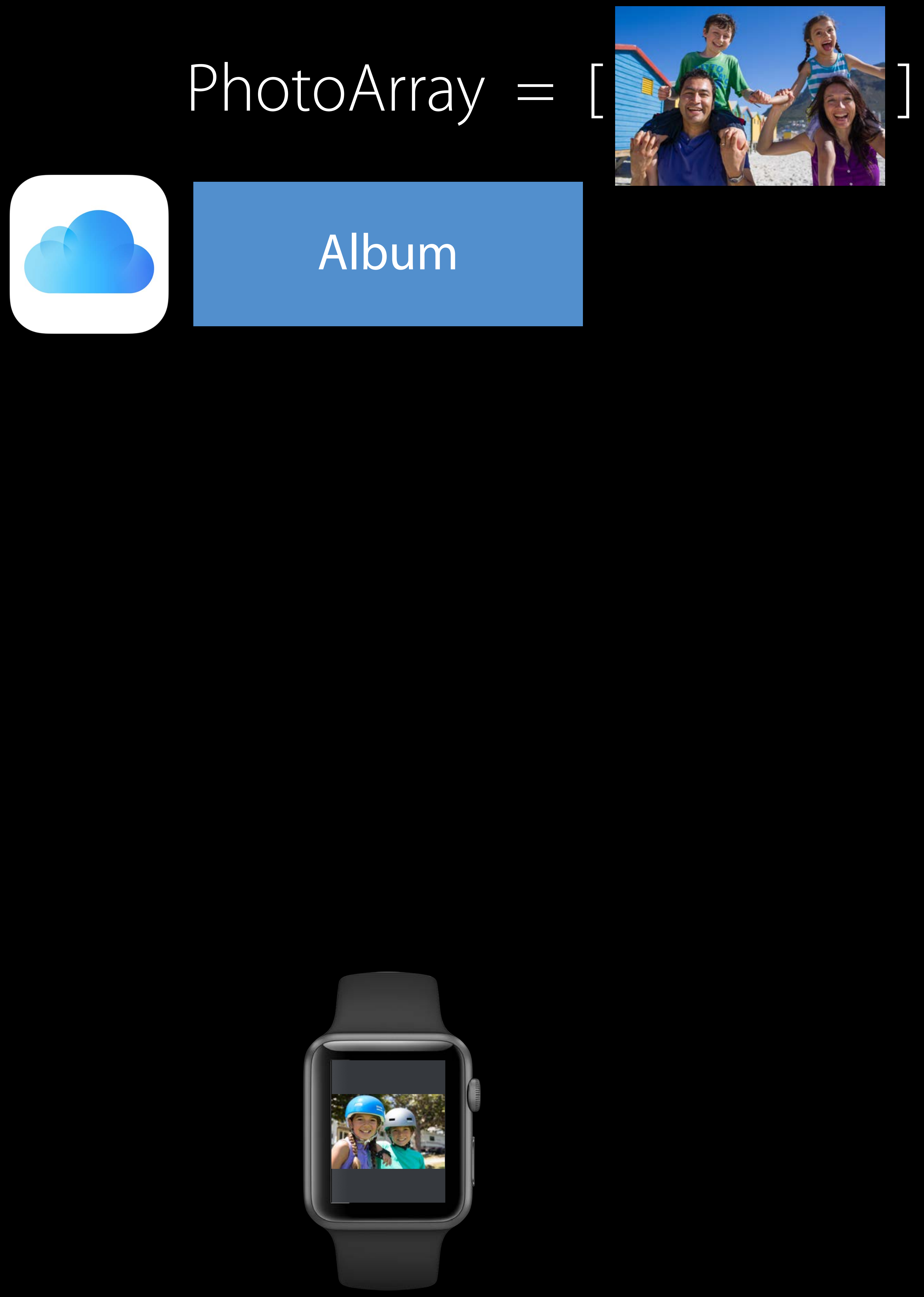
# CKReferences



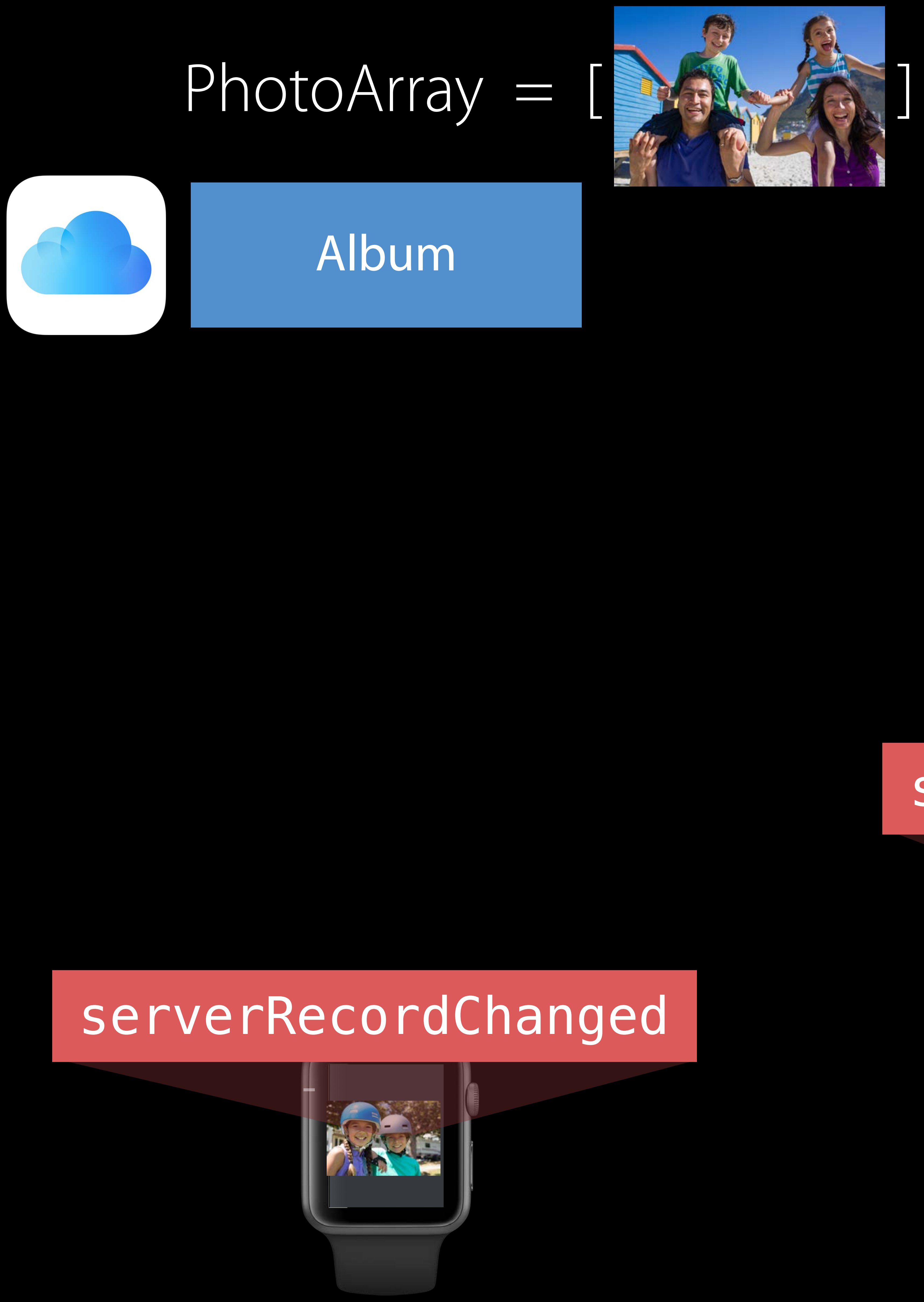
# CKReferences



# CKReferences

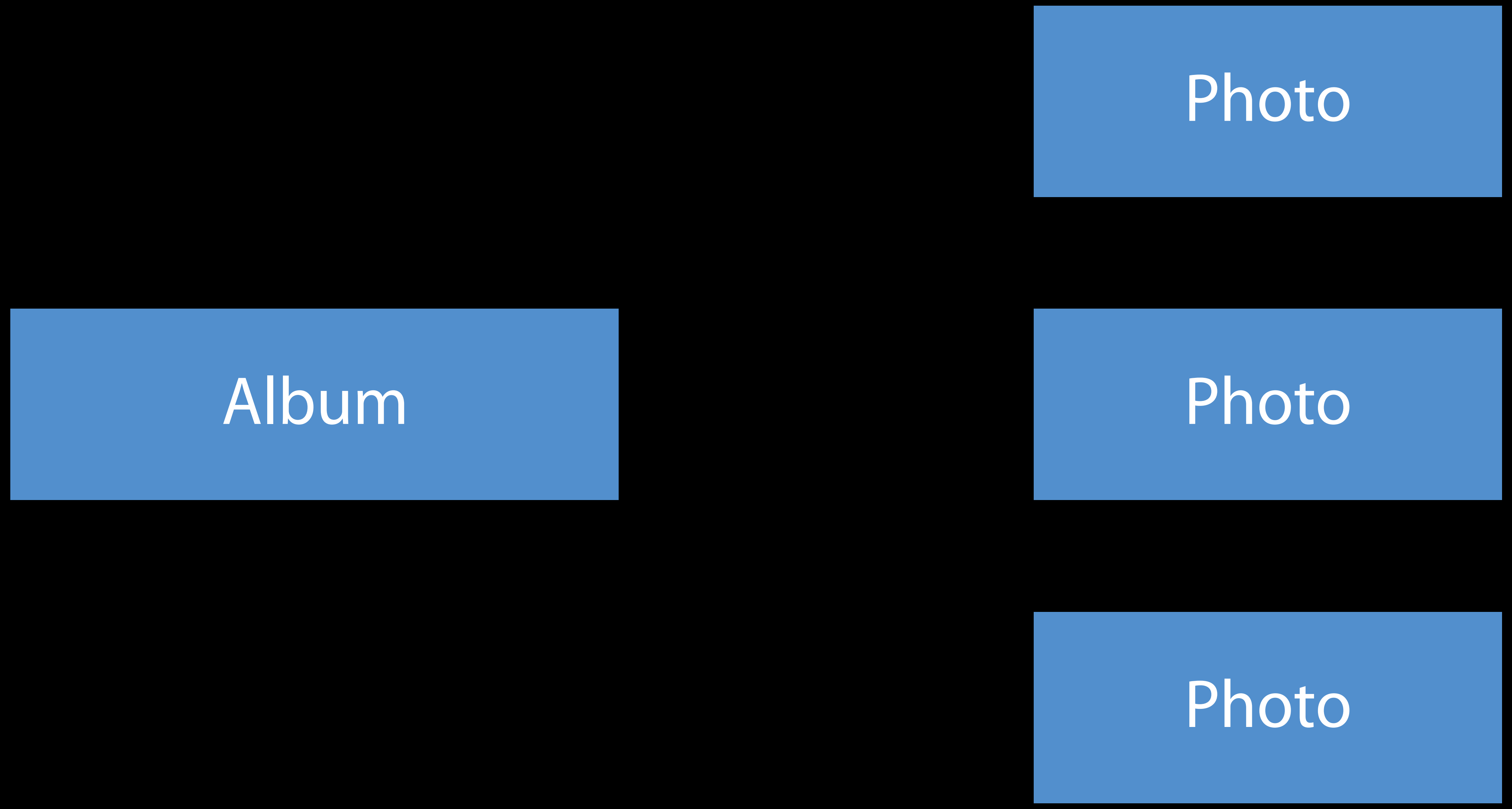


# CKReferences



## CKReferences

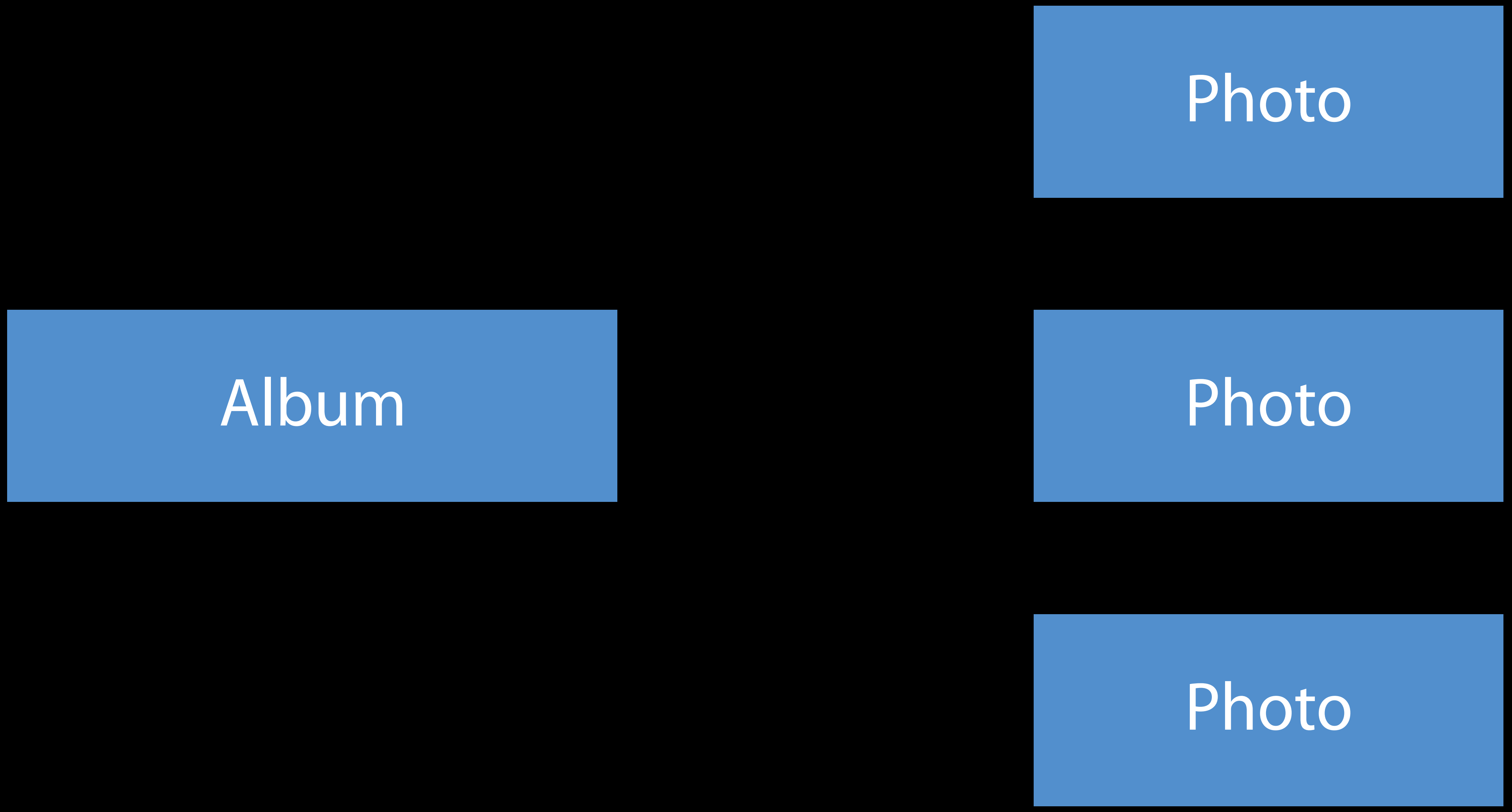
Frequent writes



# CKReferences

## Frequent writes

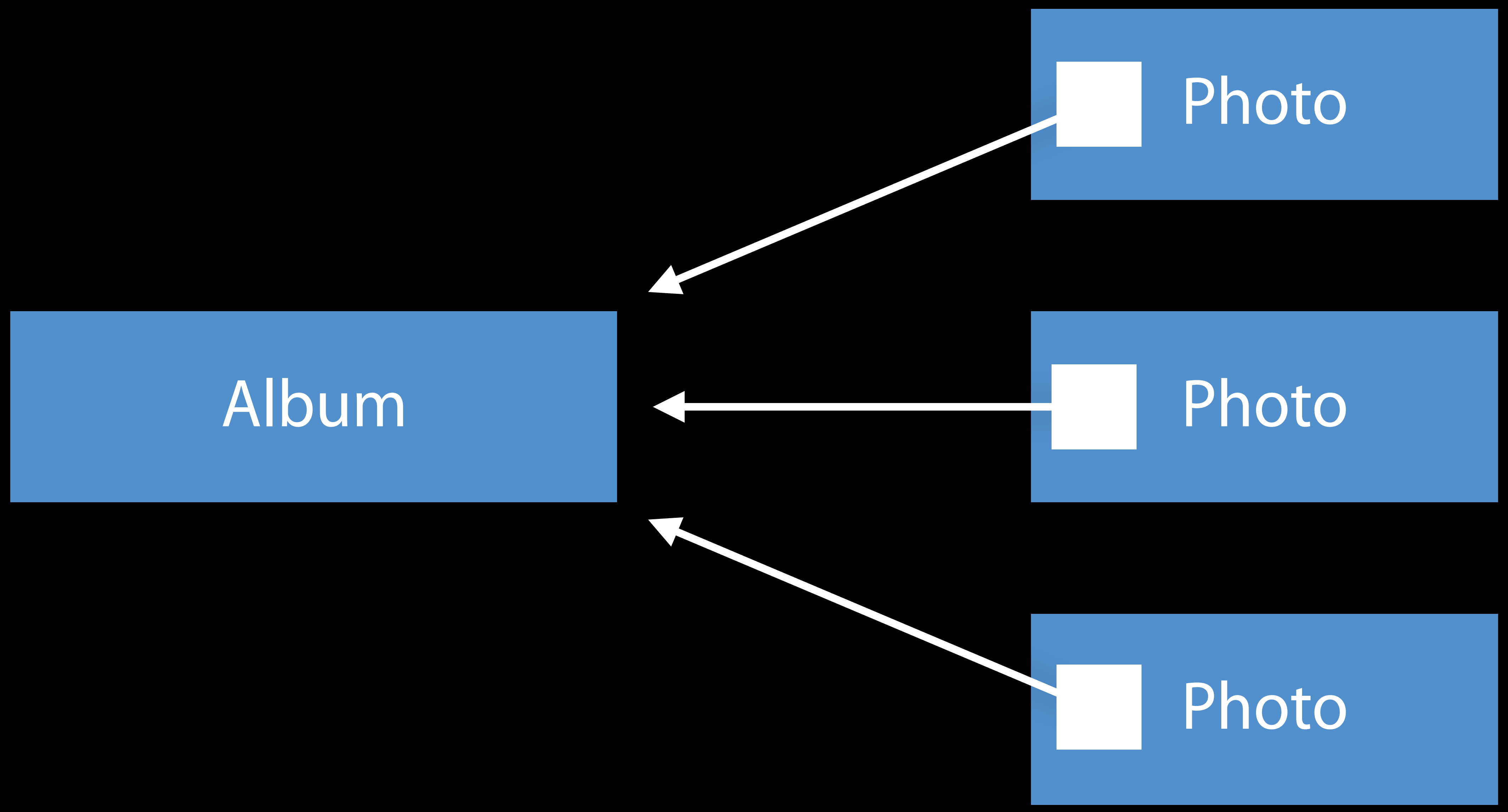
✓ Back pointers



## CKReferences

Frequent writes

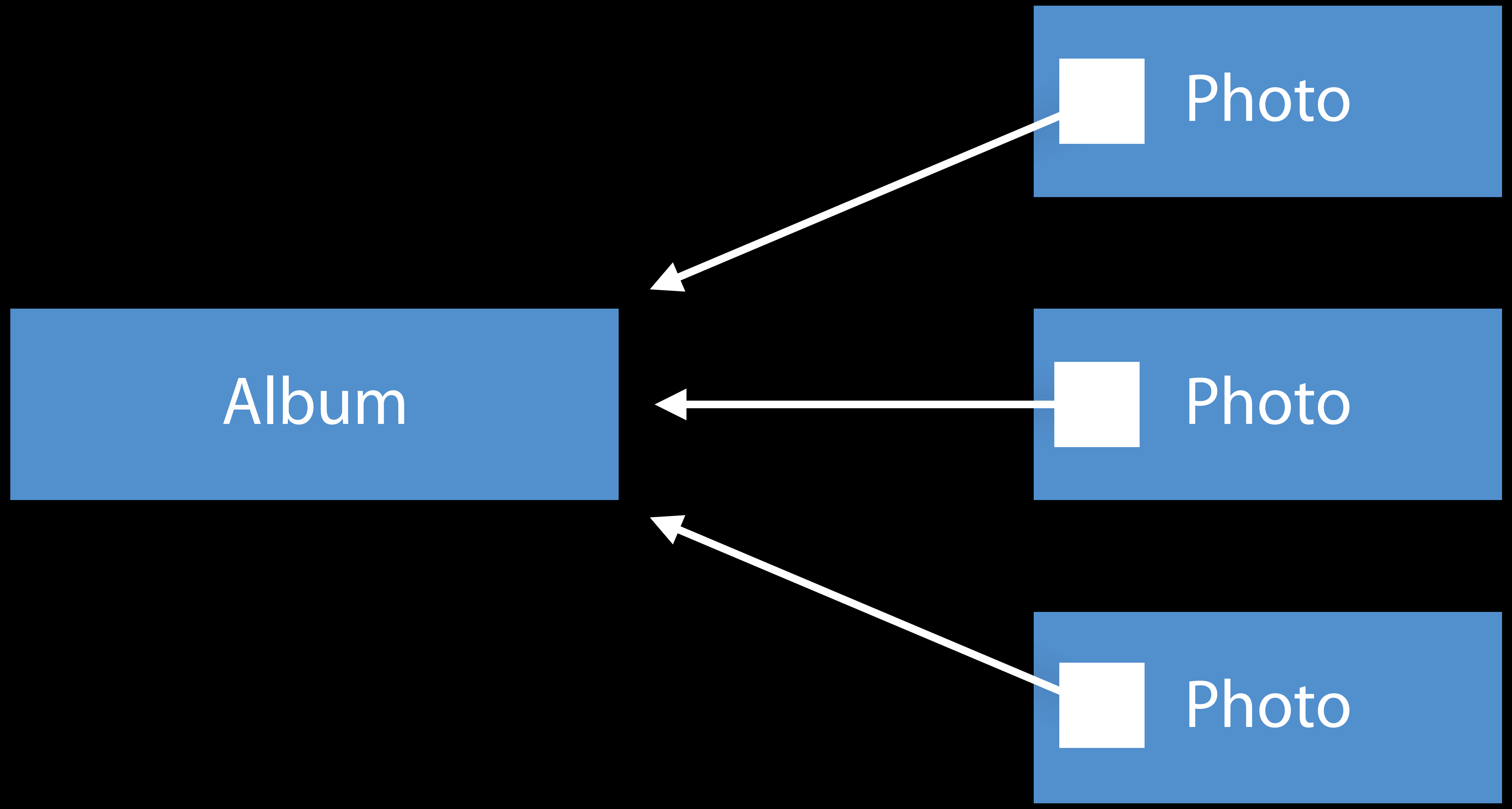
✓ Back pointers



# CKReferences

## Frequent writes

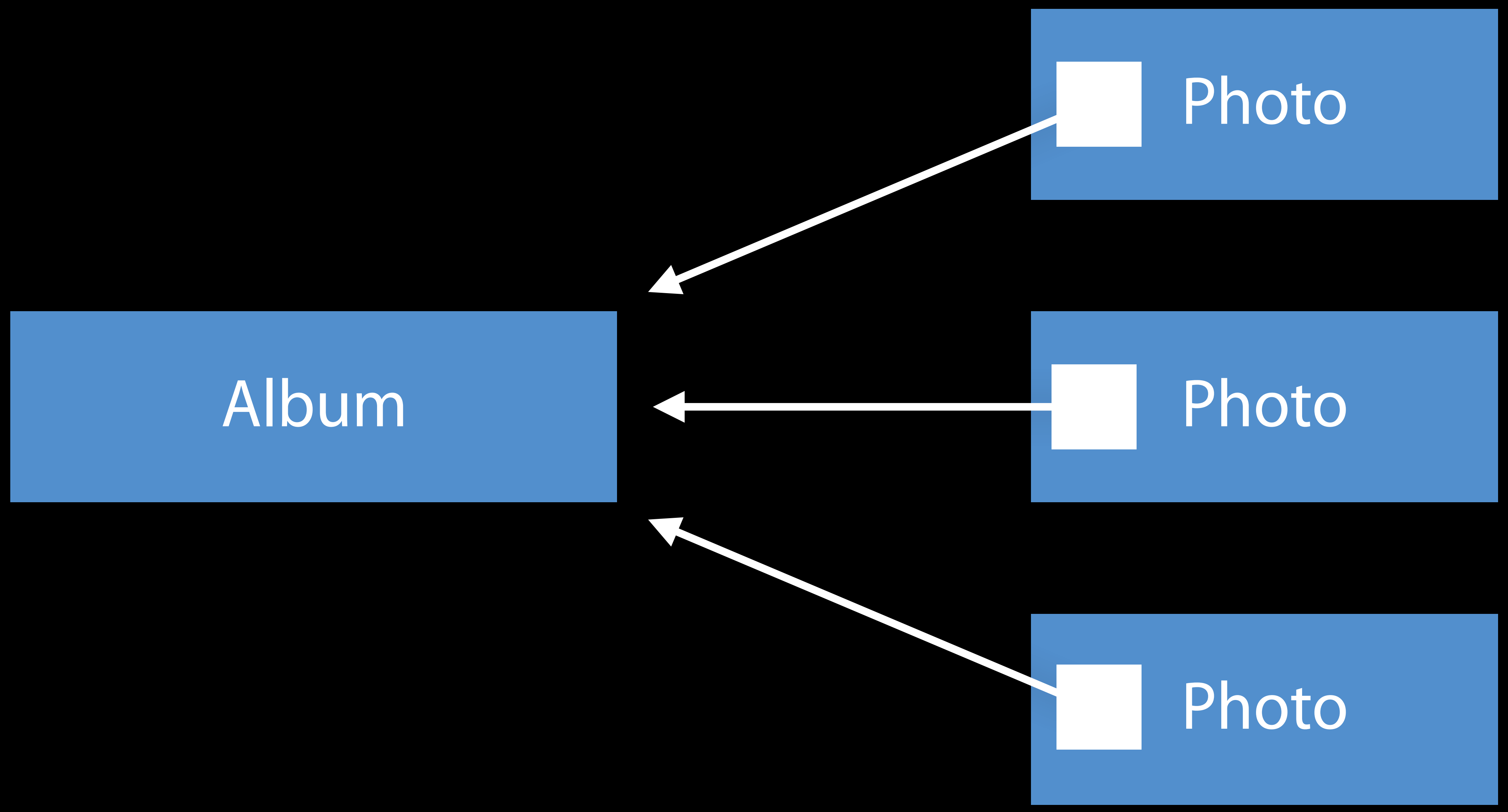
- ✓ Back pointers
- ✓ Reduce update contention



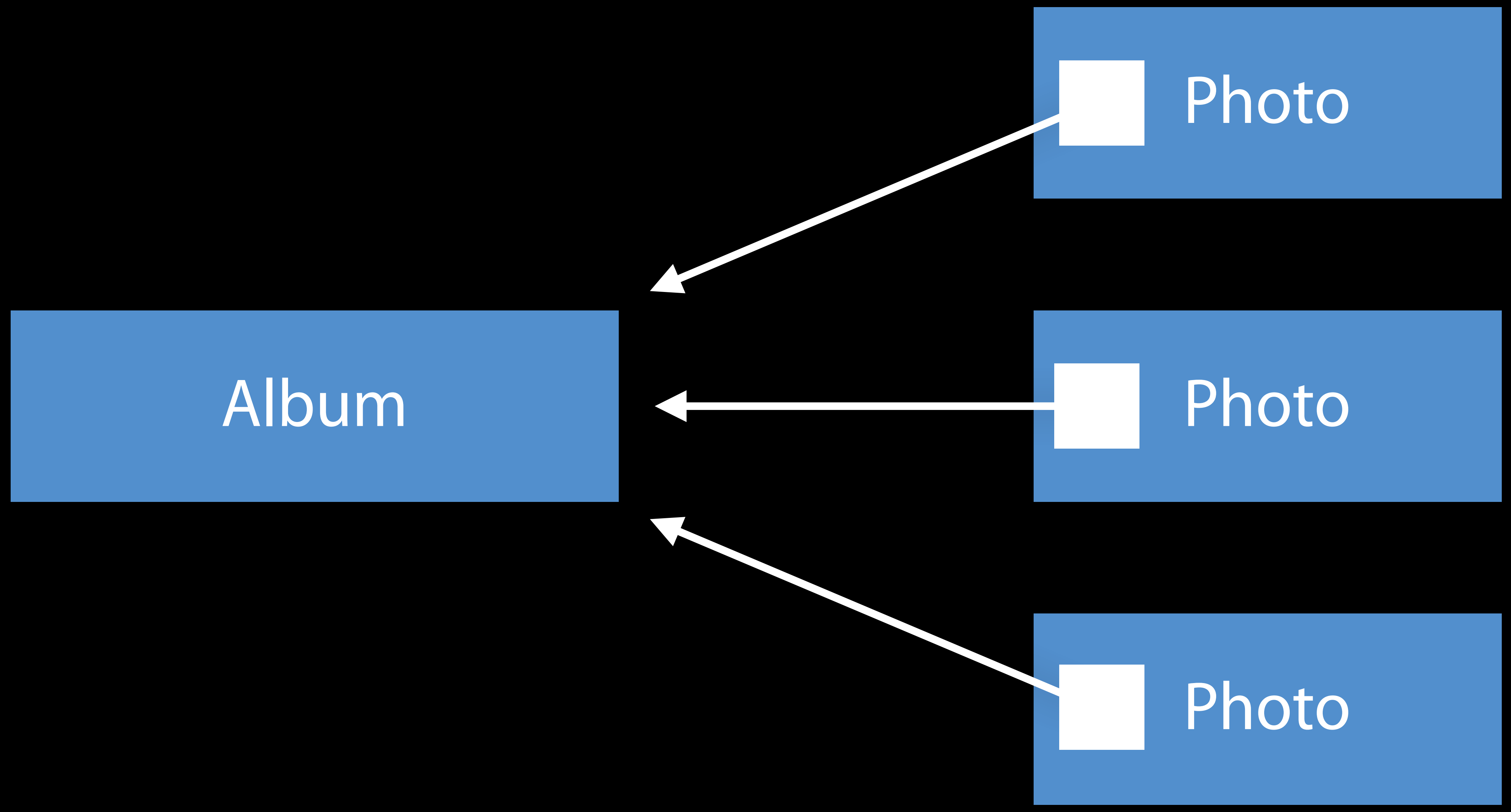
# CKReferences

## Frequent writes

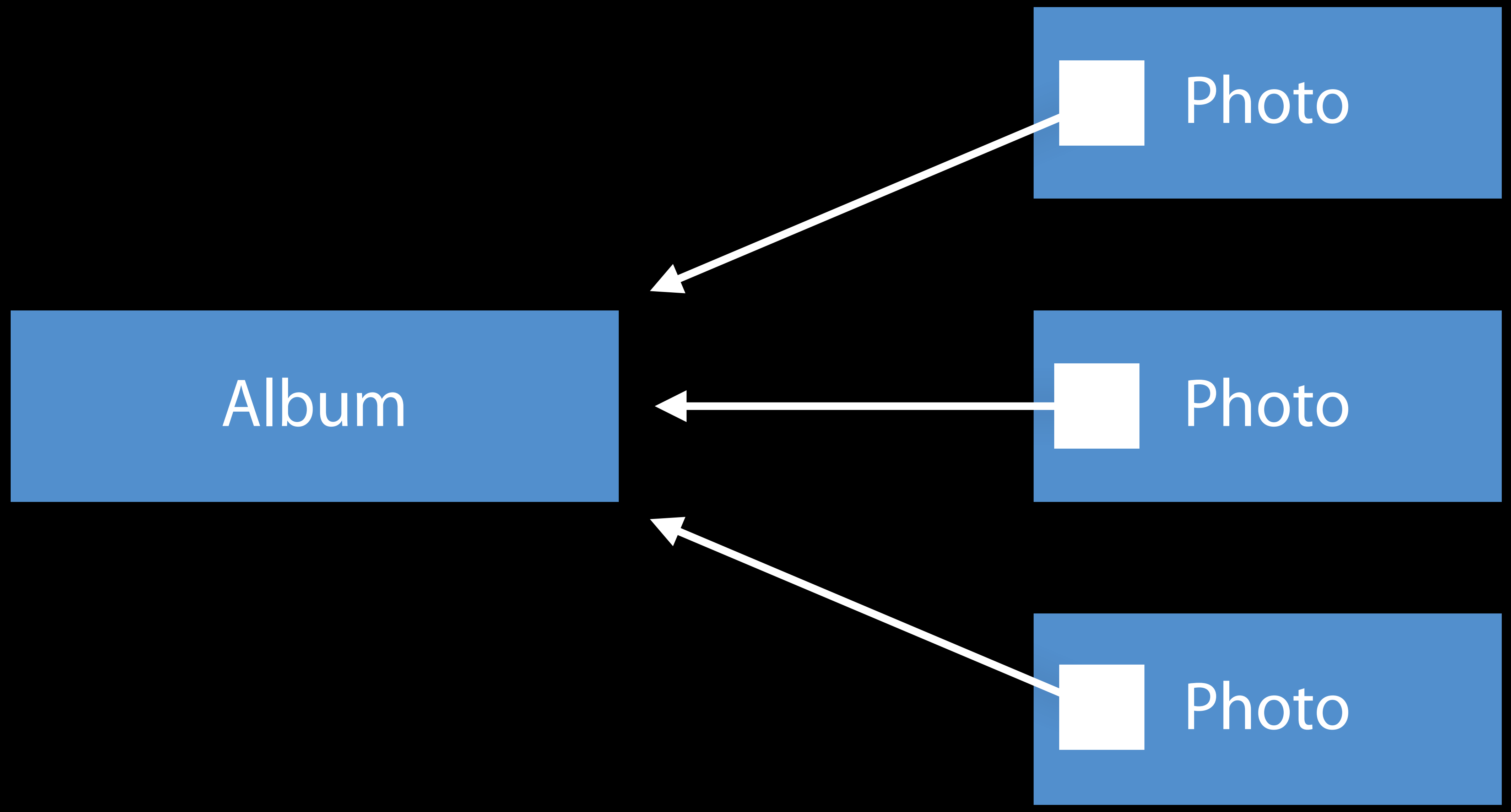
- ✓ Back pointers
- ✓ Reduce update contention
- ✓ Query for children



# CKReferences



# CKReferences



```
// Query to find all Photos in an Album
let query = CKQuery(recordType: "Photo",
   predicate: Predicate(format: "AlbumReference == %@", argumentArray: [albumRecord.recordID]))
```

## CKReferences

Parent references

## CKReferences

### Parent references

```
public class CKRecord : NSObject, NSSecureCoding, NSCopying {
   @NSCopying
   public var parent: CKReference?
}
```

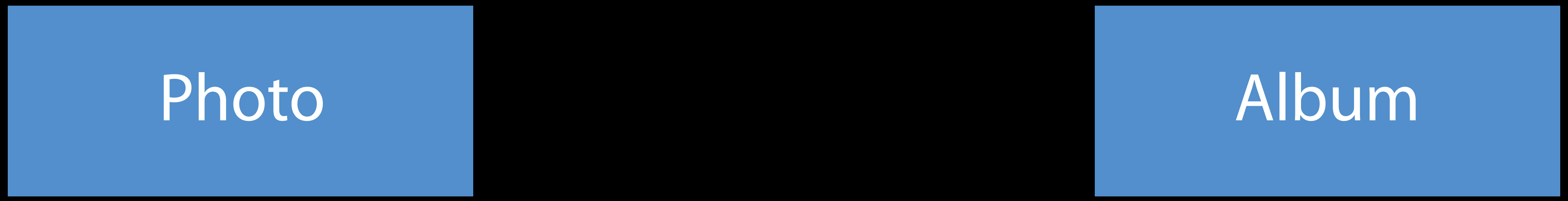
# CKReferences

## Parent references

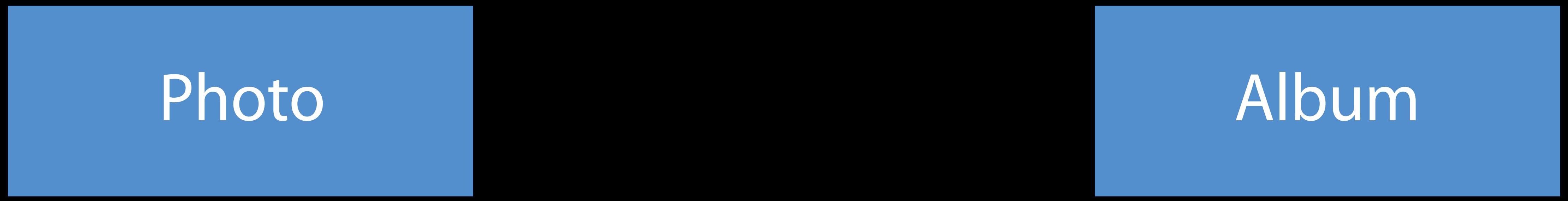
```
public class CKRecord : NSObject, NSSecureCoding, NSCopying {
   @NSCopying
   public var parent: CKReference?
}
```

Set parent references if your app supports sharing

# CKReferences

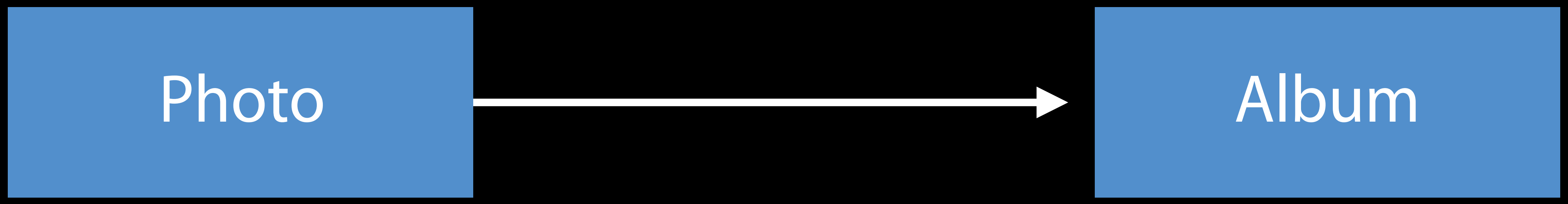


# CKReferences



```
let photoRecord = CKRecord(recordType: "Photo")
…
photoRecord.setParent(albumRecordID)
```

# CKReferences

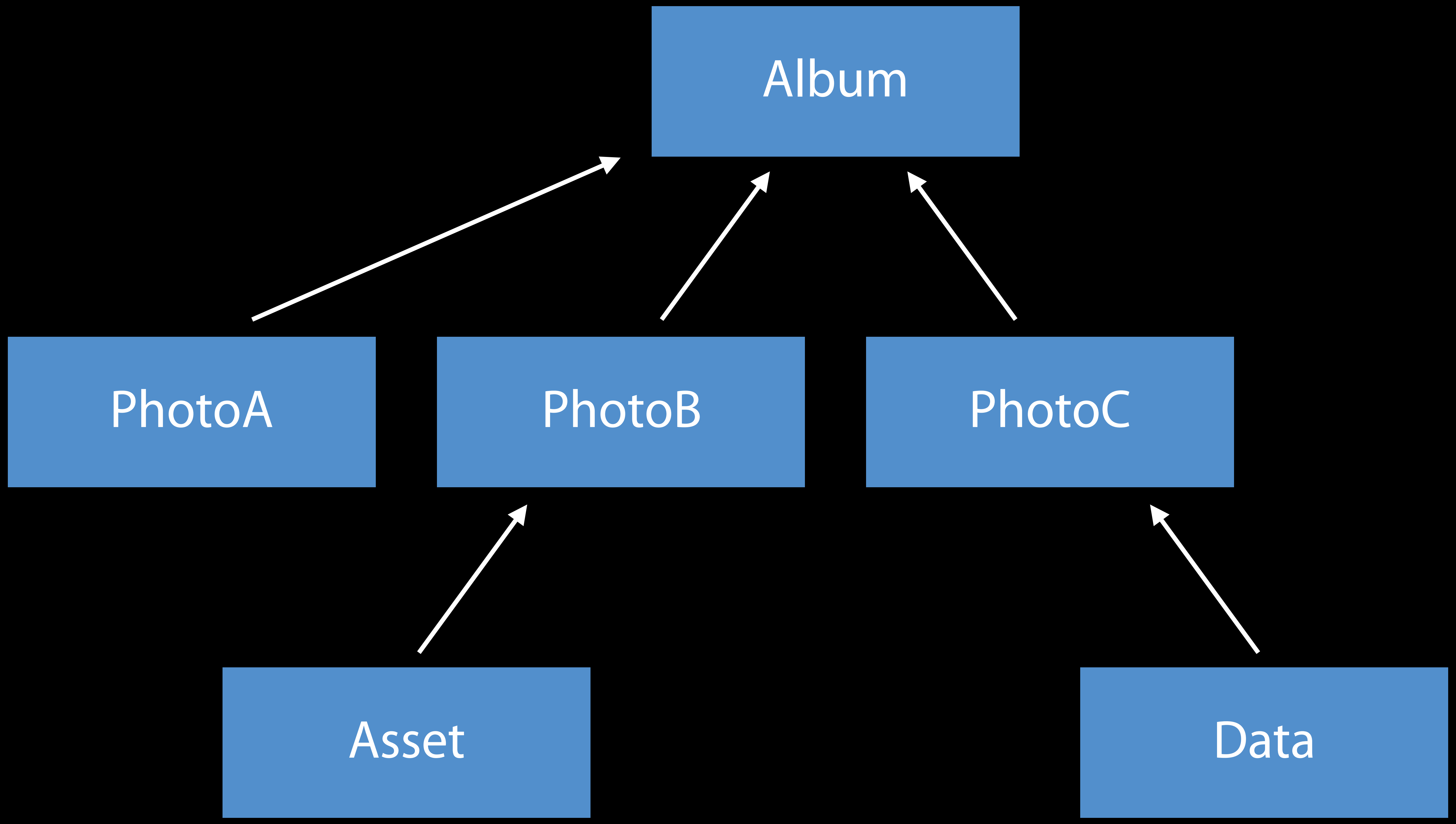


```
let photoRecord = CKRecord(recordType: "Photo")
…
photoRecord.setParent(albumRecordID)
```

# CKReferences

## Parent references

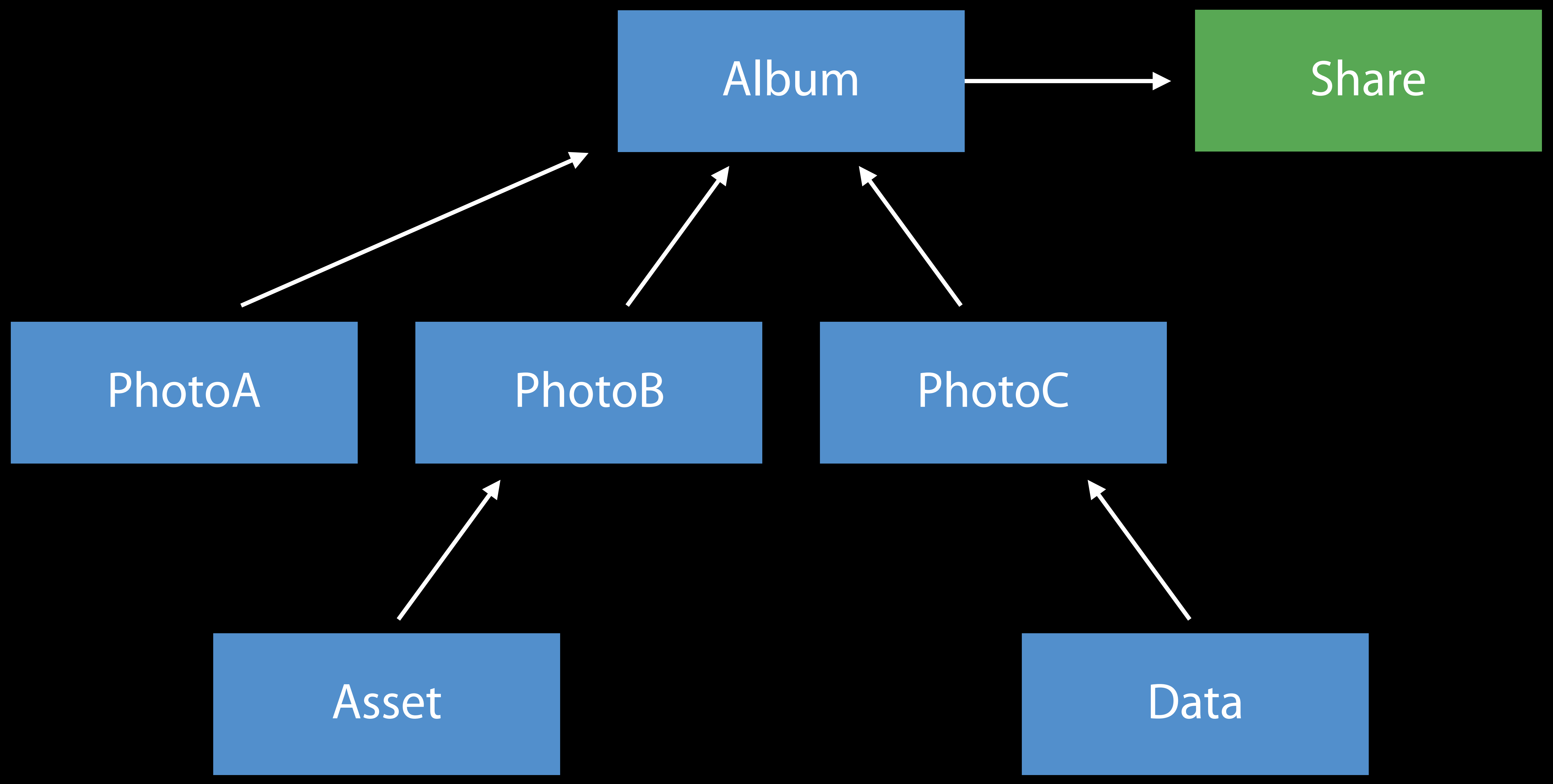
Set parent references if your app supports sharing



# CKReferences

## Parent references

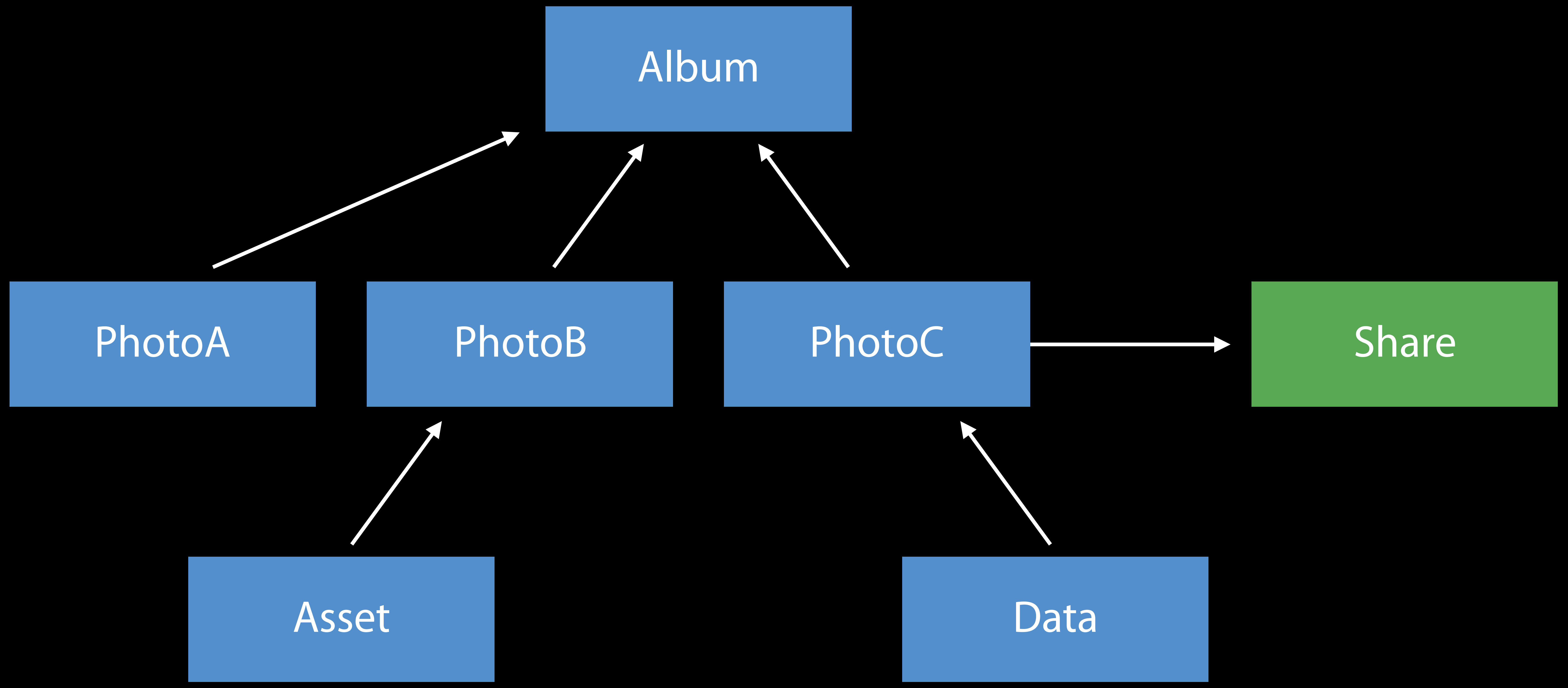
Set parent references if your app supports sharing



# CKReferences

## Parent references

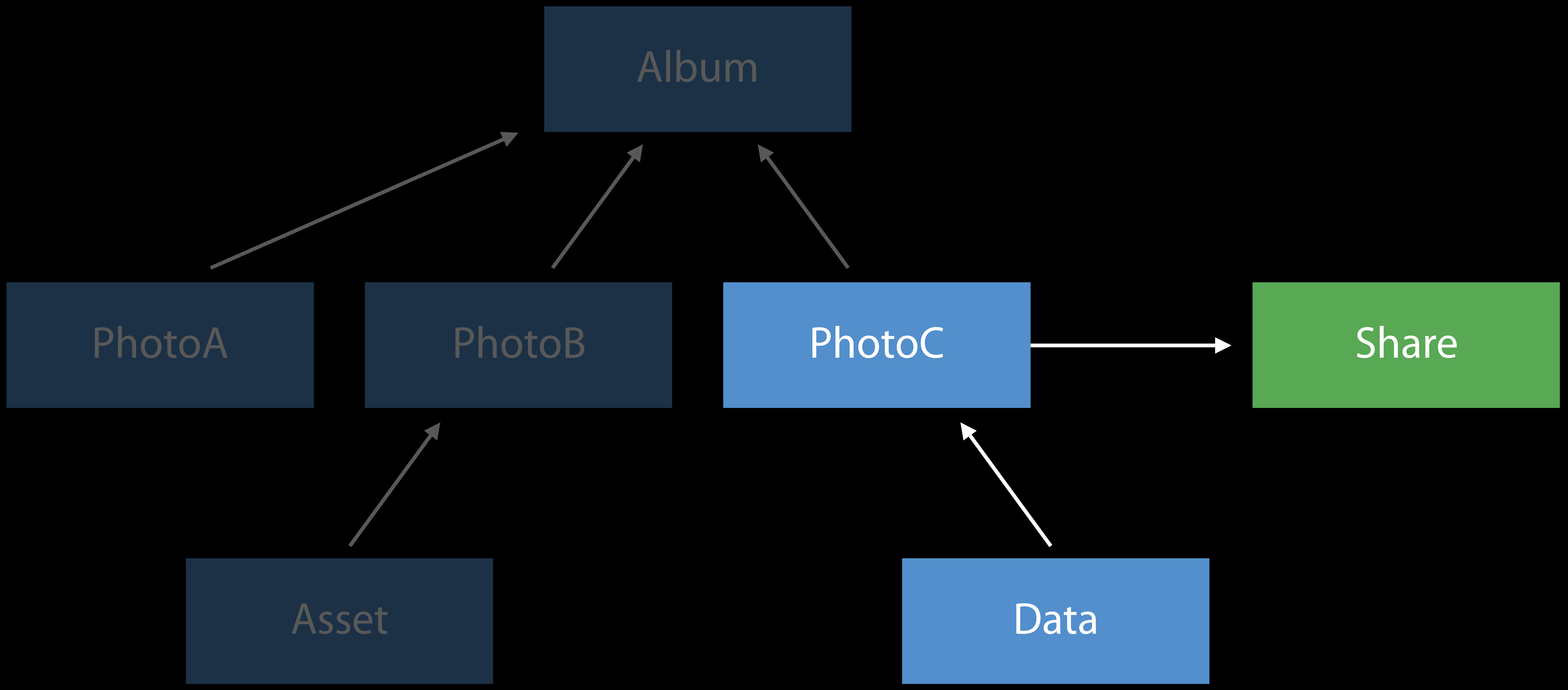
Set parent references if your app supports sharing



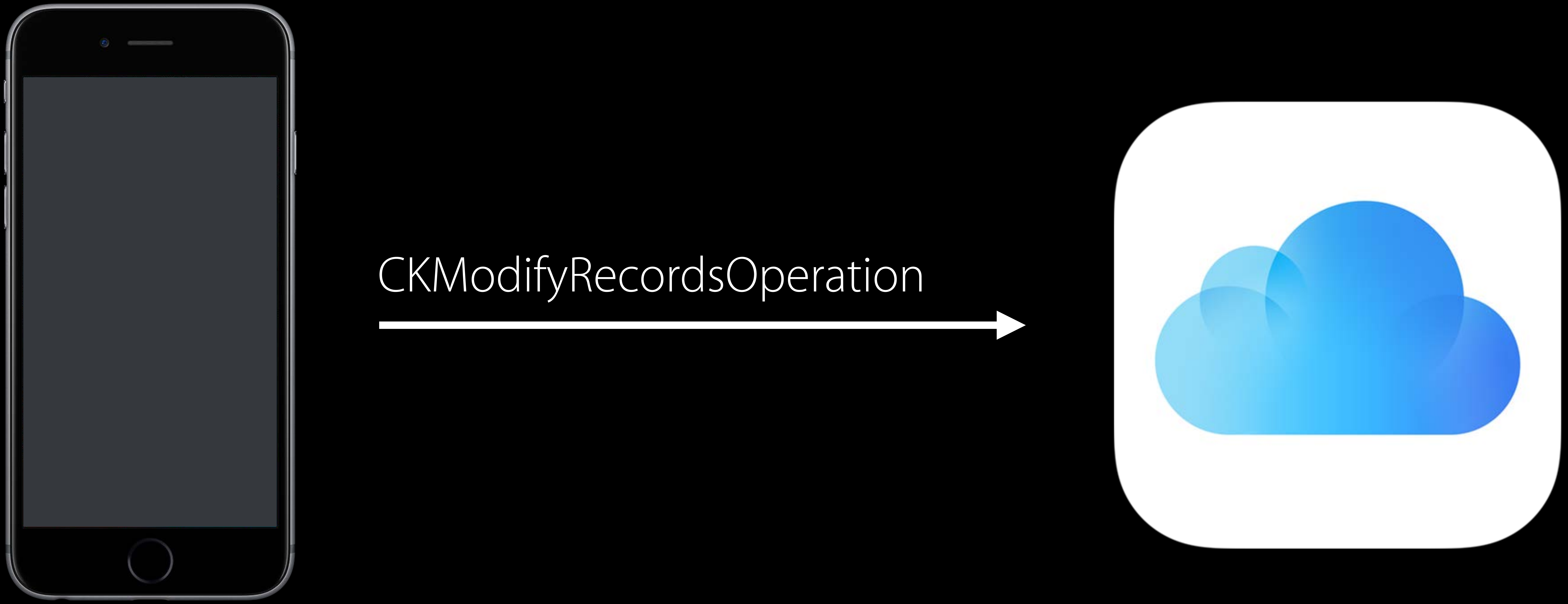
# CKReferences

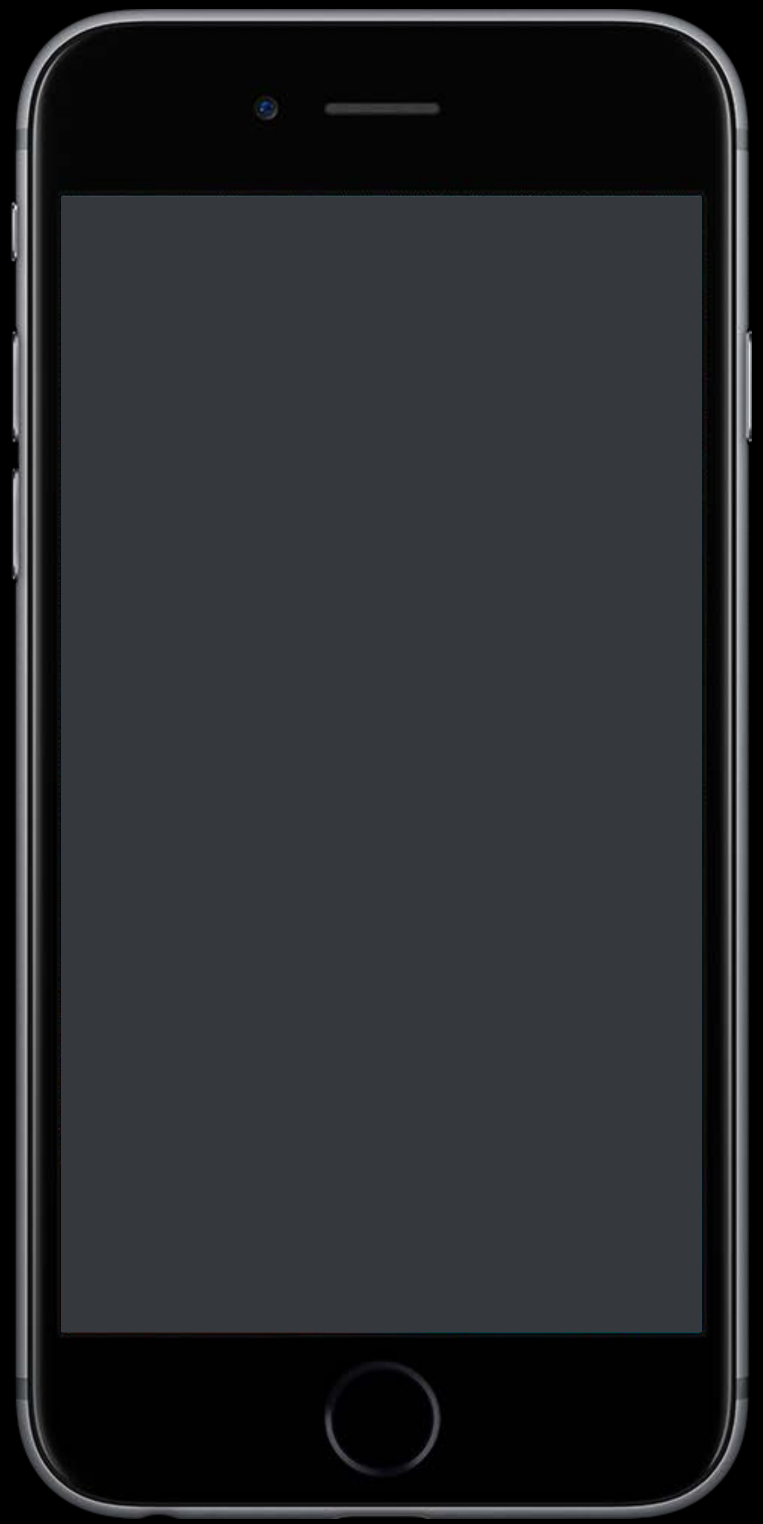
## Parent references

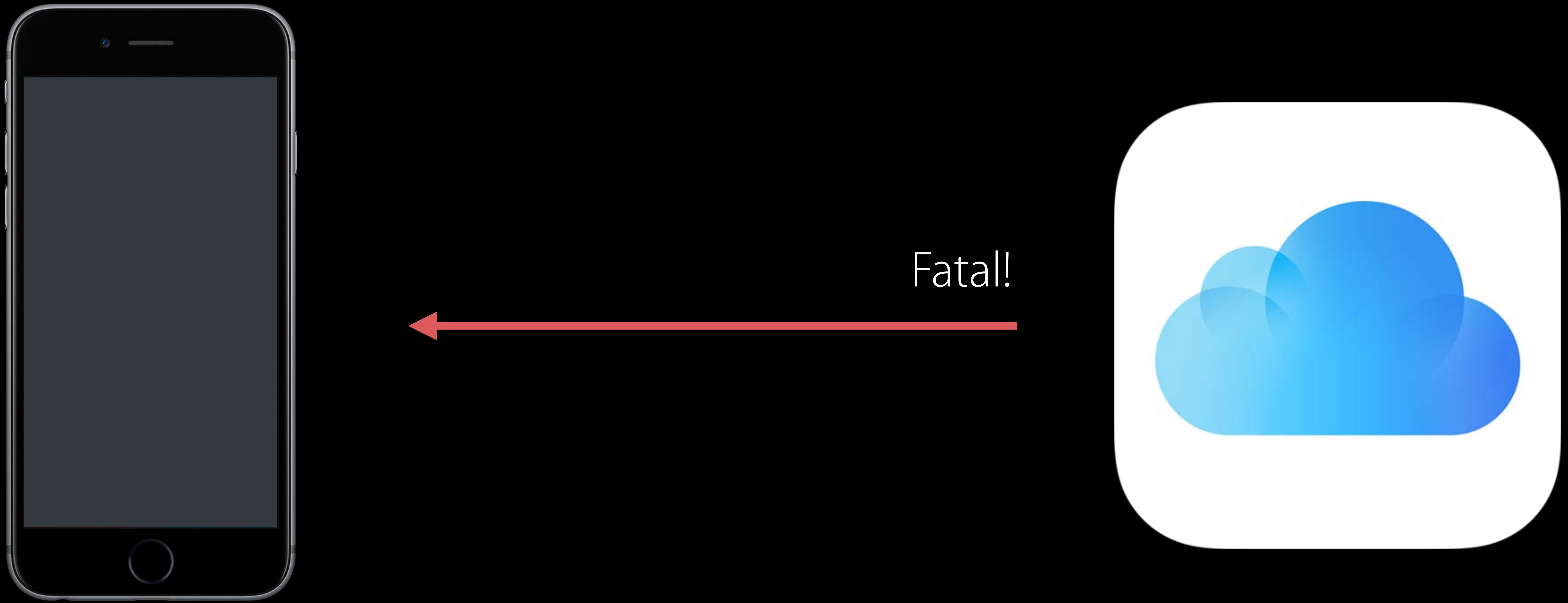
Set parent references if your app supports sharing

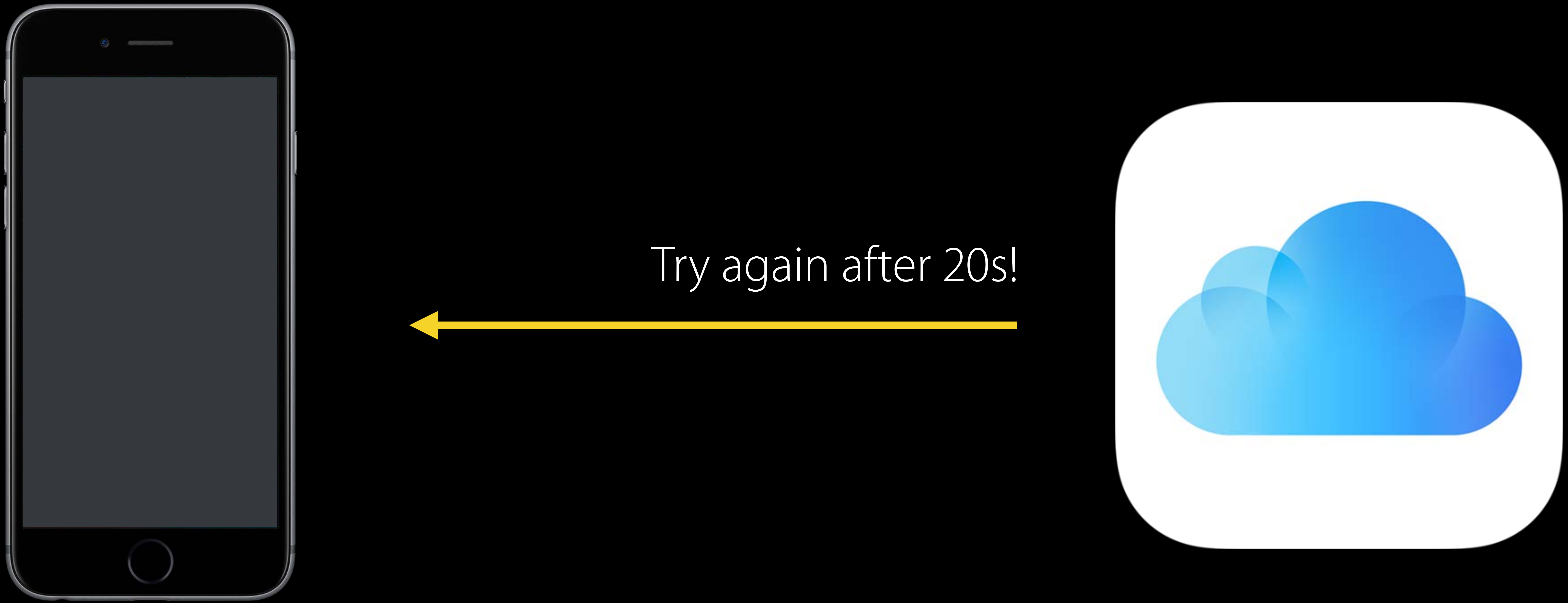


## Error Handling


CKModifyRecordsOperation




Fatal!


Try again after 20s!

## Error Handling

Fatal errors

# Error Handling

## Fatal errors

```
public enum CKErrorCode : Int {
```

## Error Handling

### Fatal errors

```
public enum CKErrorCode : Int {
    case internalError
```

## Error Handling

### Fatal errors

```
public enum CKErrorCode : Int {
    case internalError
    case serverRejectedRequest
```

## Error Handling

### Fatal errors

```
public enum CKErrorCode : Int {
    case internalError
    case serverRejectedRequest
    case invalidArguments
```

## Error Handling

### Fatal errors

```
public enum CKErrorCode : Int {
    case internalError
    case serverRejectedRequest
    case invalidArguments
    case permissionFailure
```

## Error Handling

### Fatal errors

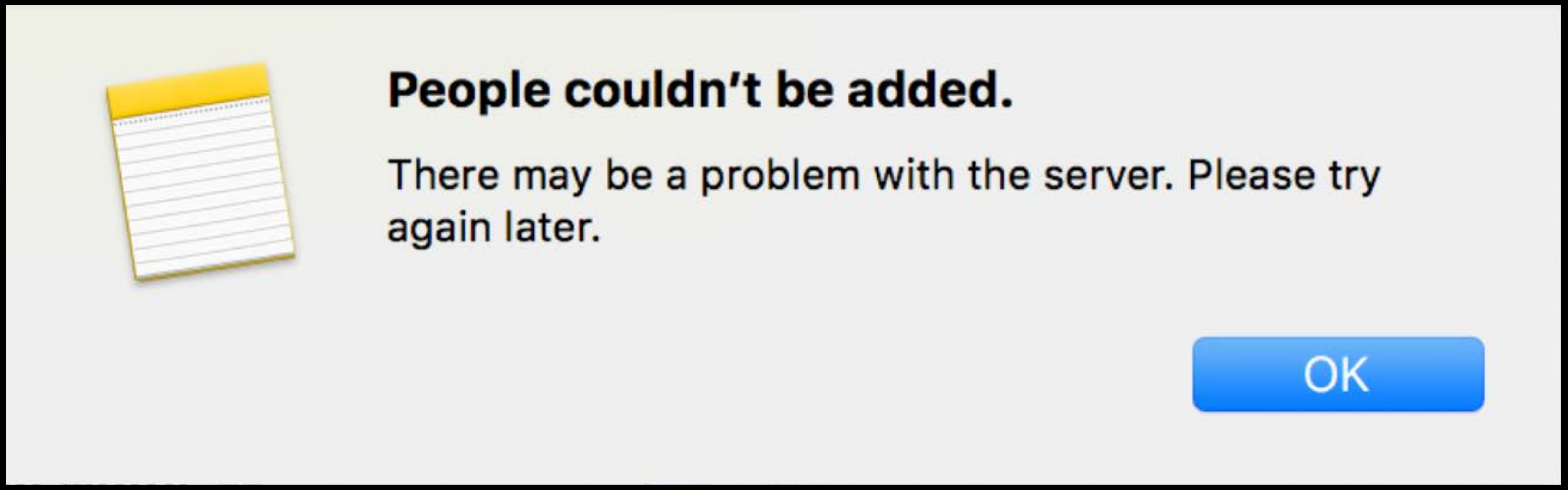
```
public enum CKErrorCode : Int {
    case internalError
    case serverRejectedRequest
    case invalidArguments
    case permissionFailure
}
```

## Error Handling

### Fatal errors

```
public enum CKErrorCode : Int {
    case internalError
    case serverRejectedRequest
    case invalidArguments
    case permissionFailure
}
```

Operations should not be retried



**People couldn't be added.**

There may be a problem with the server. Please try again later.

## Error Handling

Retry case

# Error Handling

Retry case

```
public enum CKErrorCode : Int {
```

# Error Handling

## Retry case

```
public enum CKErrorCode : Int {
    case zoneBusy
```

## Error Handling

Retry case

```
public enum CKErrorCode : Int {
    case zoneBusy
    case serviceUnavailable
```

## Error Handling

### Retry case

```
public enum CKErrorCode : Int {
    case zoneBusy
    case serviceUnavailable
    case requestRateLimited
```

## Error Handling

### Retry case

```
public enum CKErrorCode : Int {
    case zoneBusy
    case serviceUnavailable
    case requestRateLimited
}
```

## Error Handling

### Retry case

```
public enum CKErrorCode : Int {
    case zoneBusy
    case serviceUnavailable
    case requestRateLimited
}
```

Implement application-level retry using `CKErrorRetryAfterKey`

```swift
// Using CKErrorRetryAfterKey

var error = ... // Error from the previous CKOperation

if let retryAfter = error.userInfo[CKErrorRetryAfterKey] as? Double {
    let delayTime = DispatchTime.now() + retryAfter
    DispatchQueue.main.after(when: delayTime) {
        // Initialize CKOperation for a retry
    }
}
```

## Error Handling

Unavailable states

## Error Handling

### Unavailable states

Device offline

## Error Handling

### Unavailable states

Device offline

## Error Handling

Unavailable states

Device offline

`networkUnavailable`

# Error Handling

## Unavailable states

Device offline

`networkUnavailable`

- Monitor network reachability

## Error Handling

### Unavailable states

Device offline

`networkUnavailable`

- Monitor network reachability
- `SCNetworkReachability`

## Error Handling

### Unavailable states

Device offline

`networkUnavailable`

- Monitor network reachability
- `SCNetworkReachability`
- Save changes to your local cache

## Error Handling

### Unavailable states

Account unavailable

`notAuthenticated`

## Error Handling

### Unavailable states

Account unavailable

`notAuthenticated`

```
public static let CKAccountChanged: NSNotification.Name

class CKContainer {
    public func accountStatus(completionHandler: (CKAccountStatus, NSError?) -> Void)
}
```

## Summary

## Summary

Subscribing and fetch changes to efficiently stay up to date

## Summary

Subscribing and fetch changes to efficiently stay up to date

Batch updates with CKOperation

## Summary

Subscribing and fetch changes to efficiently stay up to date

Batch updates with CKOperation

Schema design

## Summary

Subscribing and fetch changes to efficiently stay up to date

Batch updates with CKOperation

Schema design

Error handling

## Related Sessions

| | | |
|---|---|---|
| What's New with CloudKit | Presidio | Thursday 3:00PM |

## Labs

| | | |
|---|---|---|
| CloudKit and iCloud Lab | Frameworks Lab B | Friday 12:00PM |

## More Information

https://developer.apple.com/wwdc16/231

WWDC16