App Frameworks #WWDC16

Architecting for Performance on watchOS 3

Session 227

Tyler McAtee watchOS Engineer
Todd Grooms watchOS Engineer

Agenda

Agenda

2-Second Tasks

Agenda

2-Second Tasks

Design

Agenda

2-Second Tasks

Design

Case Study

2-Second Tasks

2-Second Tasks

What are they?

2-Second Tasks

What are they?

Purposeful

2-Second Tasks

What are they?

Purposeful

Quick, short, simple

2-Second Tasks

What are they?

Purposeful

Quick, short, simple

Measured from beginning to end

2-Second Tasks

Examples

2-Second Tasks

Examples

Check a notification

2-Second Tasks

Examples

Check a notification

Set a timer

2-Second Tasks

Examples

Check a notification

Set a timer

Start a workout

2-Second Tasks

Improvements

2-Second Tasks

Improvements

Complications for all

2-Second Tasks

Improvements

Complications for all

Dock

2-Second Tasks

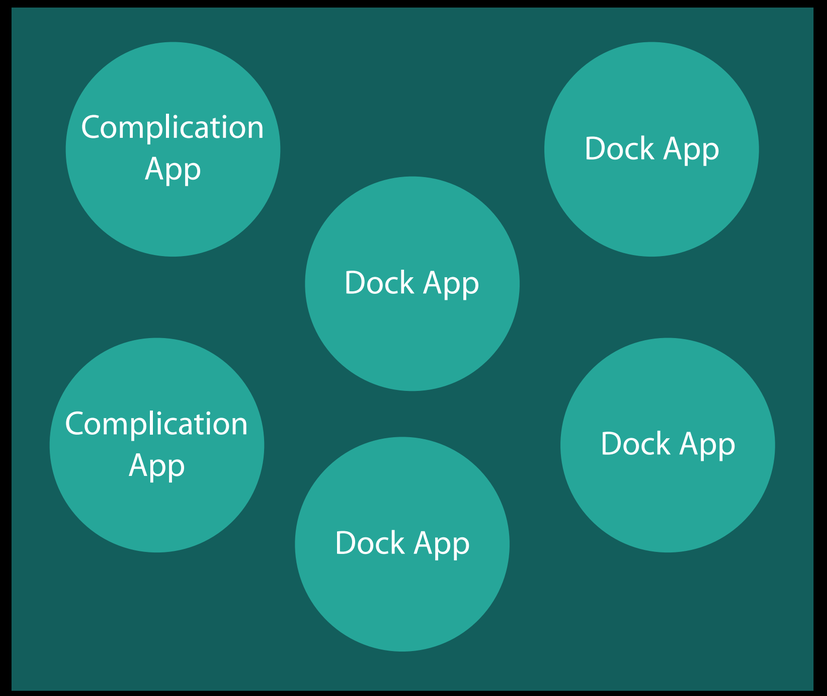
Improvements

Complications for all

Dock

2-Second Tasks

2-Second Tasks



Complication App

Dock App

Complication App

Dock App

Dock App

Dock App

Dock App

Dock App

Complication App

Dock App

Dock App

Dock App

Complication App

Dock App

Dock App

Dock App

Dock App

Dock App

Complication App

Complication App

Dock App

Dock App

Dock App

Dock App

Dock App

Complication App

Dock App

Dock App

Dock App

Dock App

Dock App

Complication App

Complication App

Process
Process
Process
Complication App
Process
Process
Process
Dock App
Complication App
Dock App
Dock App
Dock App
Dock App
Process
Workout Session
Process
Mail/ Calendar Sync
Complication App
Dock App
Process
Background Task
Dock App
Dock App
Dock App
Process
Process
Complication App
System App
Complication App
Process


10:09
12:00–1:00PM
Lunch with Graham
Home

Memory

Limit

Memory

Limit

WatchKit applications have a fixed memory limit

Memory

Limit

WatchKit applications have a fixed memory limit

Should be nowhere near the limit

Memory

Limit

WatchKit applications have a fixed memory limit

Should be nowhere near the limit

Current limit is 30MB

Memory

Strategies

Memory

Strategies

Use appropriately sized images

Memory

Strategies

Use appropriately sized images

Use appropriately sized data sets

Memory

Strategies

Use appropriately sized images

Use appropriately sized data sets

Use lightweight APIs

Memory

Strategies

Use appropriately sized images

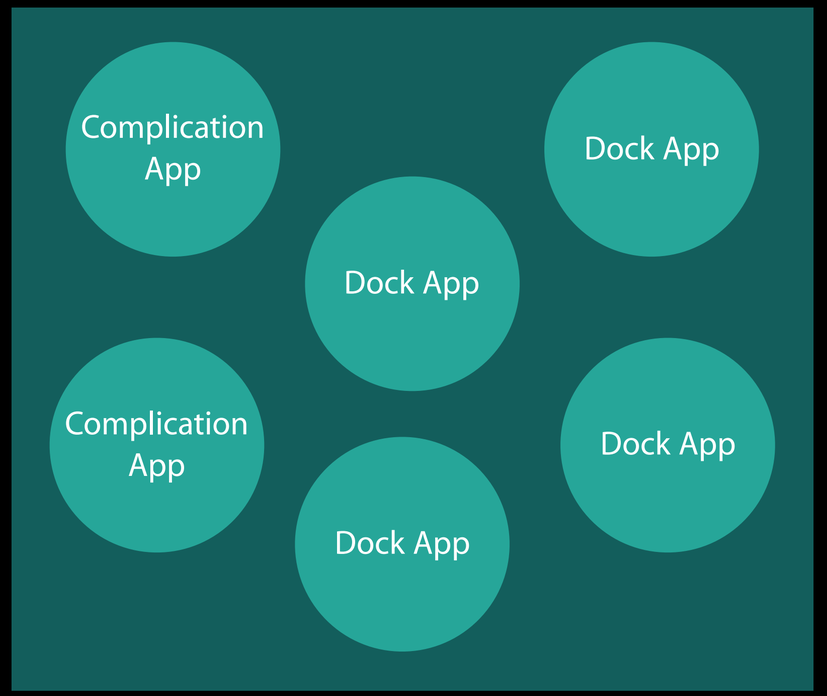
Use appropriately sized data sets

Use lightweight APIs

Don't keep around things you no longer need

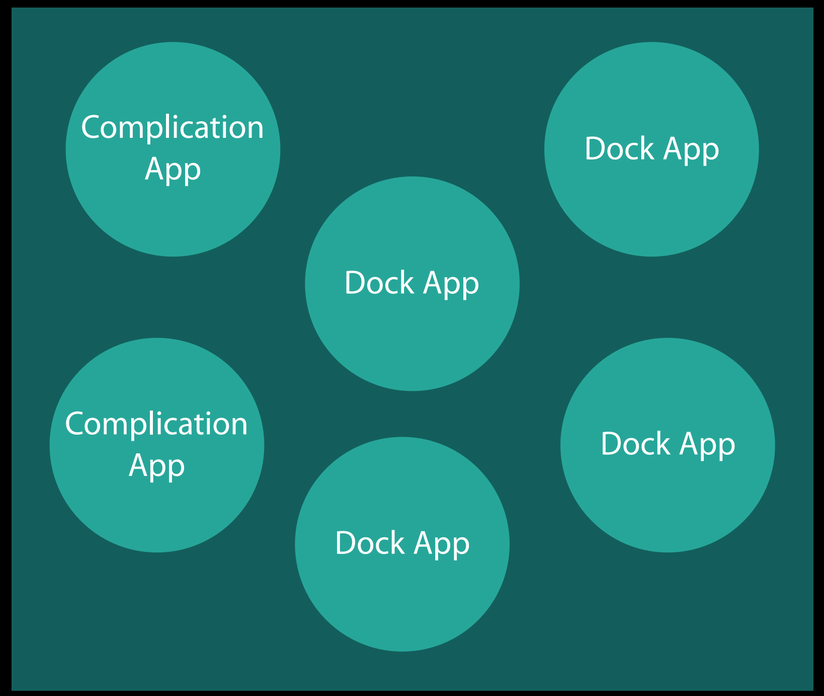

10:09
12:00–1:00PM
Lunch with Graham
Home


10:09
12:00–1:00PM
Lunch with Graham
Home


Complication App
Dock App
Dock App
Complication App
Dock App
Dock App

Resume Time

Key Path in watchOS 3



Resume Time

Key Path in watchOS 3

Resume Time

Resuming often in the Dock

Resume Time

Resuming often in the Dock

Resume Time

Lifecycle extension delegate methods

Resume Time

Lifecycle extension delegate methods

applicationDidFinishLaunching

Resume Time

Lifecycle extension delegate methods

applicationDidFinishLaunching

applicationDidBecomeActive

Resume Time

Lifecycle extension delegate methods

applicationDidFinishLaunching

applicationDidBecomeActive

applicationWillResignActive

Resume Time

Lifecycle extension delegate methods

applicationDidFinishLaunching

applicationDidBecomeActive

applicationWillResignActive

applicationDidEnterBackground

Resume Time

Lifecycle extension delegate methods

applicationDidFinishLaunching

applicationDidBecomeActive

applicationWillResignActive

applicationDidEnterBackground

applicationWillEnterForeground

Resume Time

Lifecycle interface controller methods

Resume Time

Lifecycle interface controller methods

awakeWithContext:

Resume Time

Lifecycle interface controller methods

awakeWithContext:

willActivate

Resume Time

Lifecycle interface controller methods

awakeWithContext:

willActivate

didAppear

Resume Time

Lifecycle interface controller methods

awakeWithContext:

willActivate

didAppear

Resume Time

Lifecycle interface controller methods

awakeWithContext:

willActivate

didAppear

Resume Time

Lifecycle interface controller methods

awakeWithContext:

willActivate

didAppear

willDisappear

Resume Time

Lifecycle interface controller methods

awakeWithContext:

willActivate

didAppear

willDisappear

didDeactivate

Resume Time

Lifecycle interface controller methods

awakeWithContext:

willActivate

didAppear

willDisappear

didDeactivate

Resume Time

Lifecycle interface controller methods

awakeWithContext:

willActivate

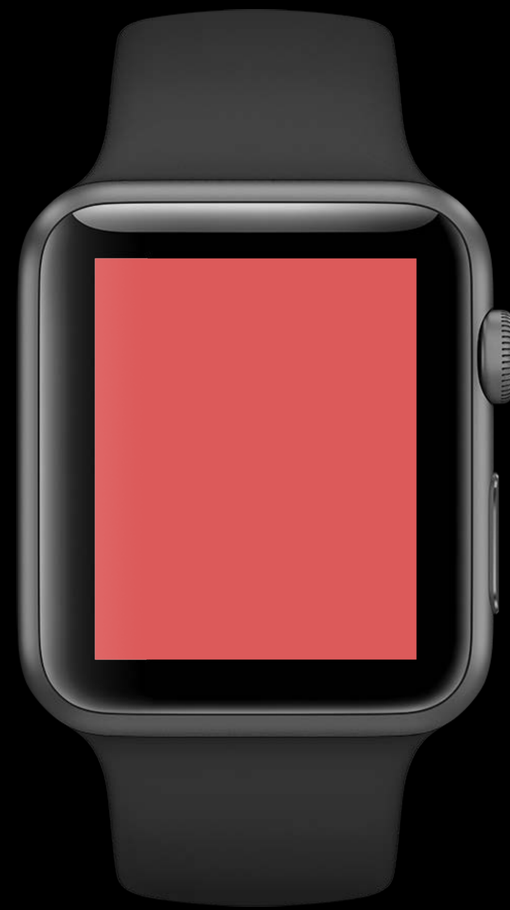
didAppear

willDisappear

didDeactivate

applicationDidFinishLaunching

applicationDidBecomeActive



awakeWithContext:

willActivate

didAppear

applicationWillResignActive

willDisappear

didDeactivate

applicationDidEnterBackground

Snapshot!

Keeping Your Watch App Up to Date

Mission

Thursday 9:00AM

Snapshot!

willActivate

didAppear

Keeping Your Watch App Up to Date

Mission

Thursday 9:00AM

handleBackgroundTasks:

Snapshot!

Snapshot!

willDisappear

didDeactivate

Keeping Your Watch App Up to Date

Mission

Thursday 9:00AM

applicationWillEnterForeground

willActivate

didAppear

applicationDidBecomeActive

Resume Time

Tips

Resume Time

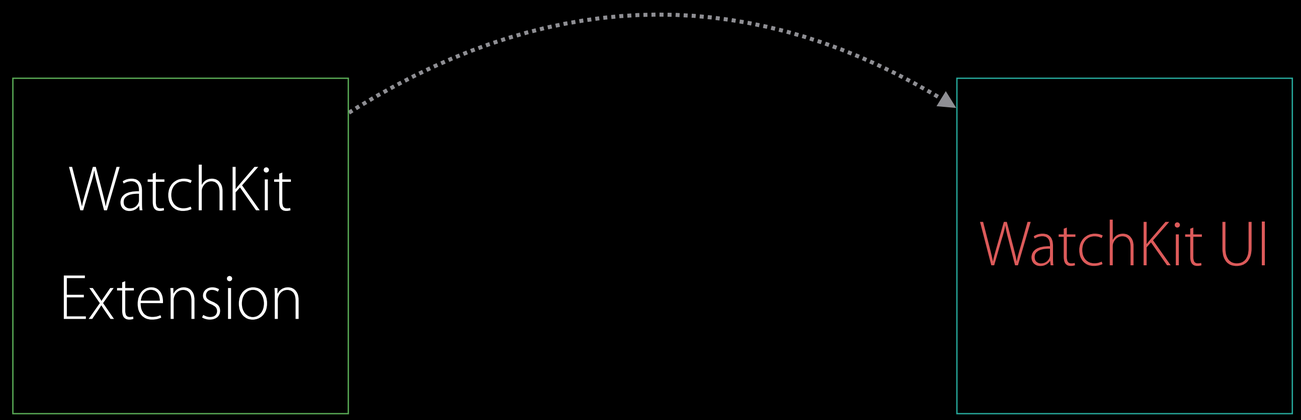
Tips

Use discretion when updating `WKInterface` objects

Resume Time

Tips

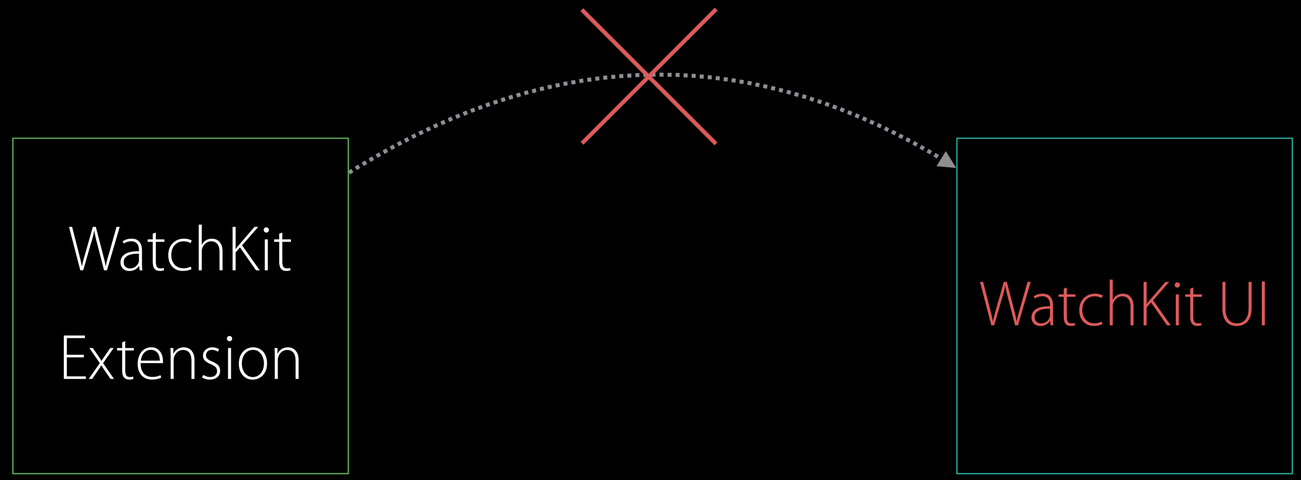
Use discretion when updating `WKInterface` objects



Resume Time

Tips

Use discretion when updating `WKInterface` objects



Resume Time

Tips

Use discretion when updating `WKInterface` objects

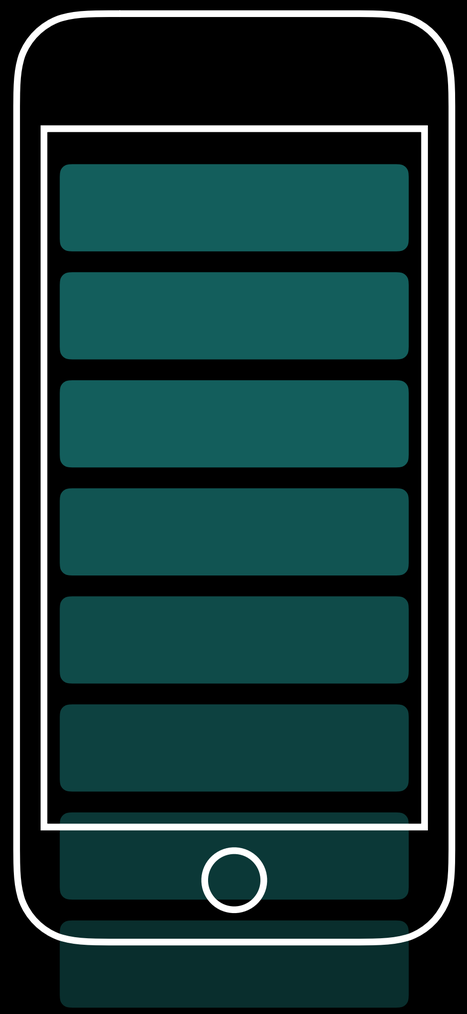
Resume Time

Tips

Resume Time

Tips

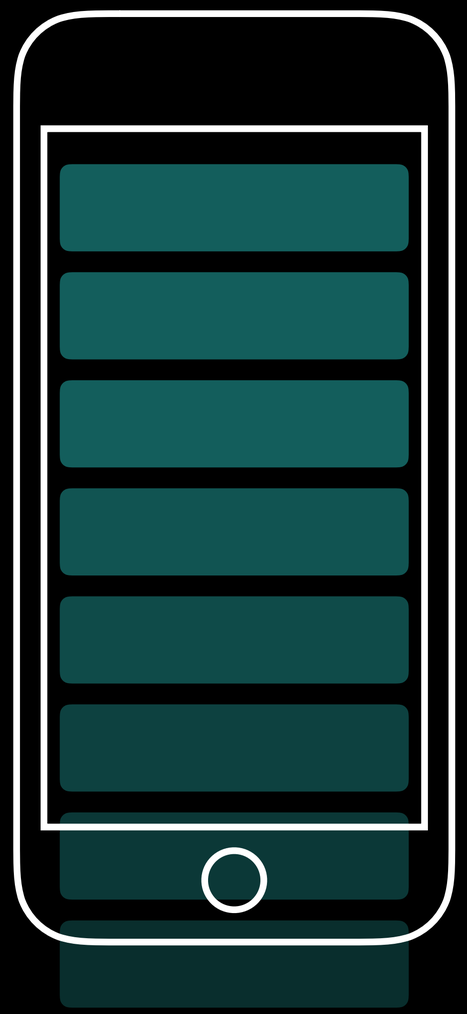
`WKInterfaceTable` is not a `UITableView`



Resume Time

Tips

`WKInterfaceTable` is not a `UITableView`



Resume Time

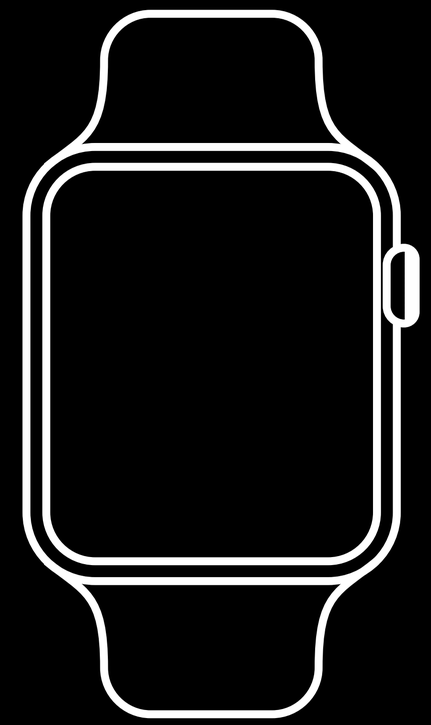
Tips

`WKInterfaceTable` is not a `UITableView`

Resume Time

Tips

`WKInterfaceTable` is not a `UITableView`



Resume Time

Tips

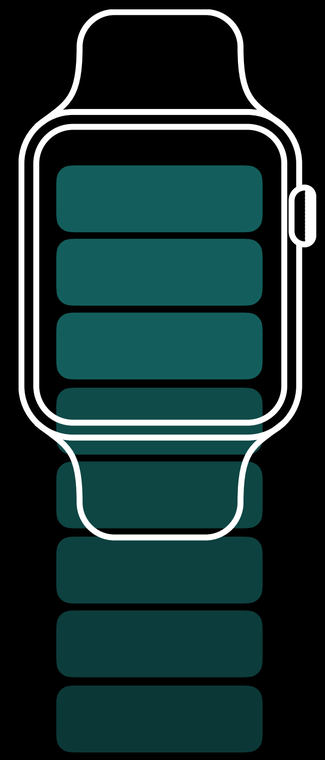
`WKInterfaceTable` is not a `UITableView`

Resume Time

Tips

`WKInterfaceTable` is not a `UITableView`

Keep `WKInterfaceTable` size down



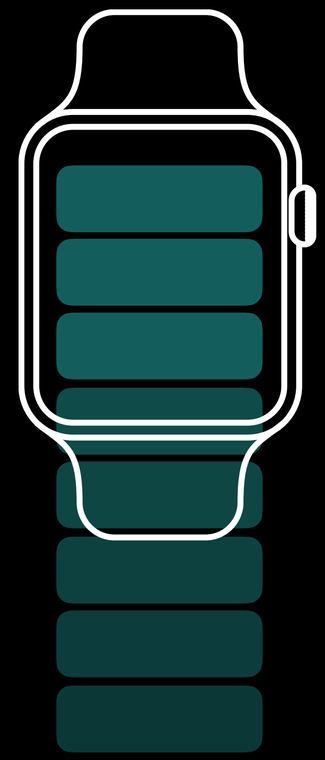
Resume Time

Tips

`WKInterfaceTable` is not a `UITableView`

Keep `WKInterfaceTable` size down

Avoid reloading a `WKInterfaceTable` when possible



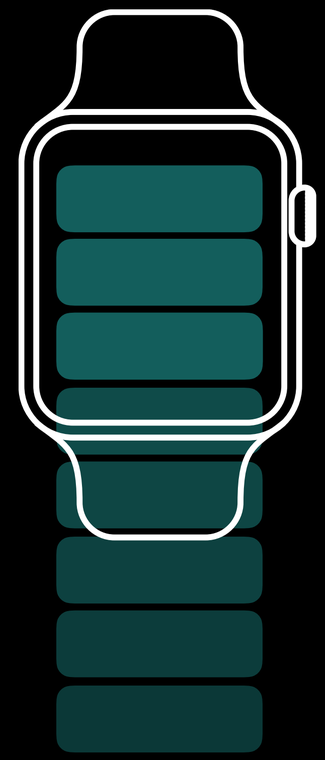
Resume Time

Tips

`WKInterfaceTable` is not a `UITableView`

Keep `WKInterfaceTable` size down

Avoid reloading a `WKInterfaceTable` when possible



Resume Time

Tips

`WKInterfaceTable` is not a `UITableView`

Keep `WKInterfaceTable` size down

Avoid reloading a `WKInterfaceTable` when possible

Design

Design

Design

Glanceable UI

Design

Glanceable UI

Focused purpose

Design

Glanceable UI

Focused purpose

Navigation

Design

Glanceable UI

Focused purpose

Navigation

Design

Glanceable UI

Focused purpose

Navigation

Design

Glanceable UI

Focused purpose

Navigation

Design

Glanceable UI

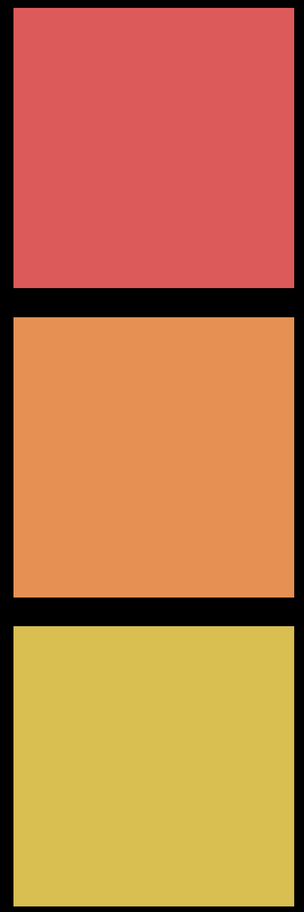
Focused purpose

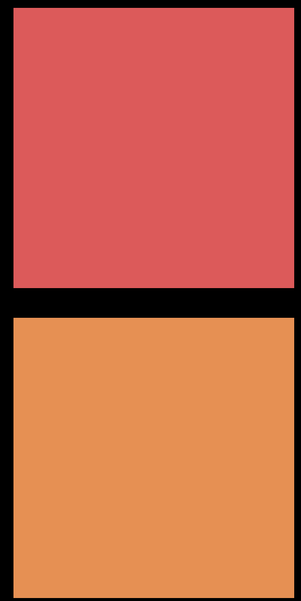
Navigation

Quick Interaction Techniques for watchOS Presidio Wednesday 11:00AM

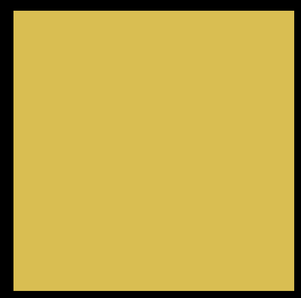
contextForSegue:withIdentifier:inTable:

contextForSegue:withIdentifier:inTable:

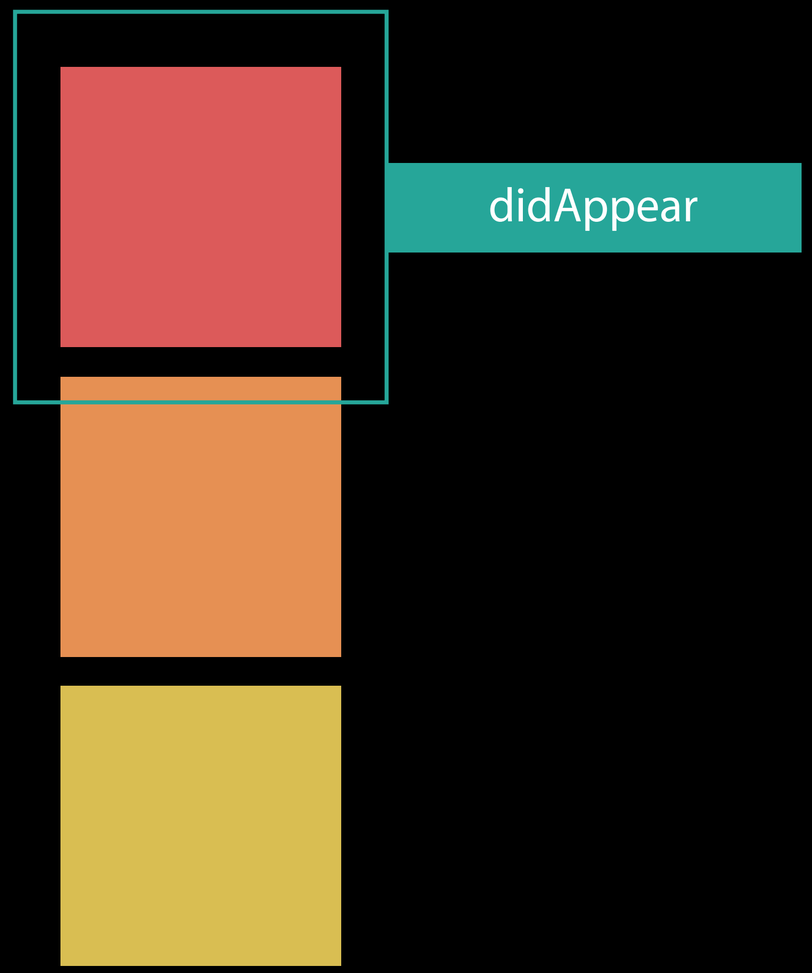


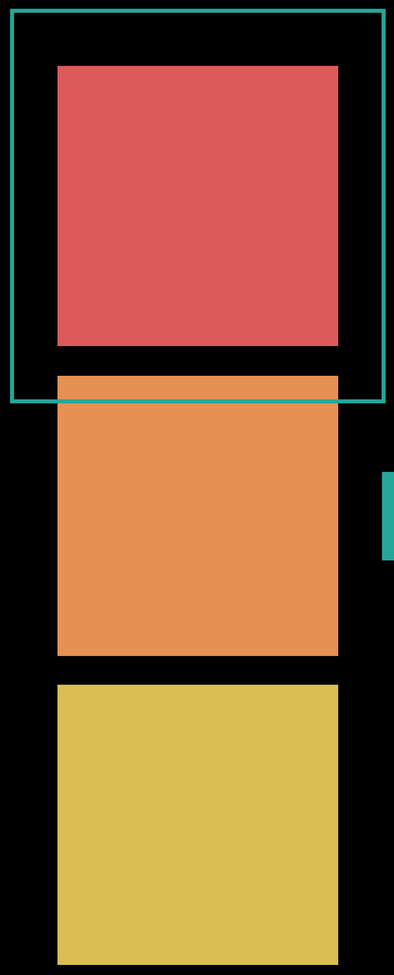


awakeWithContext

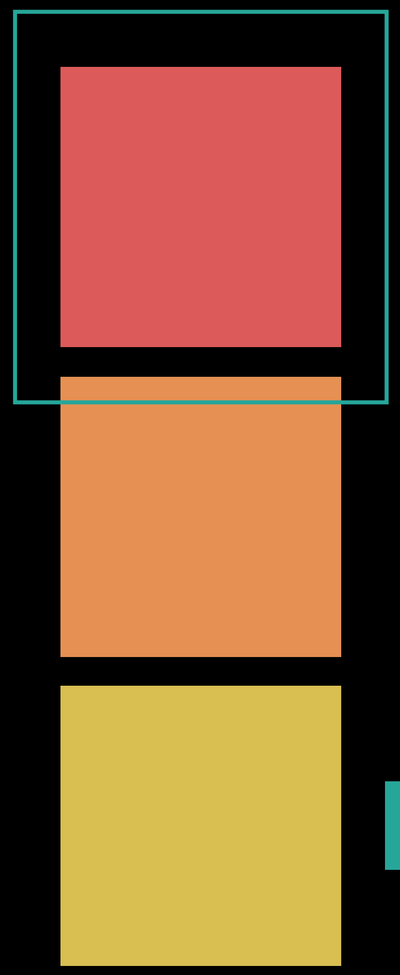


willActivate

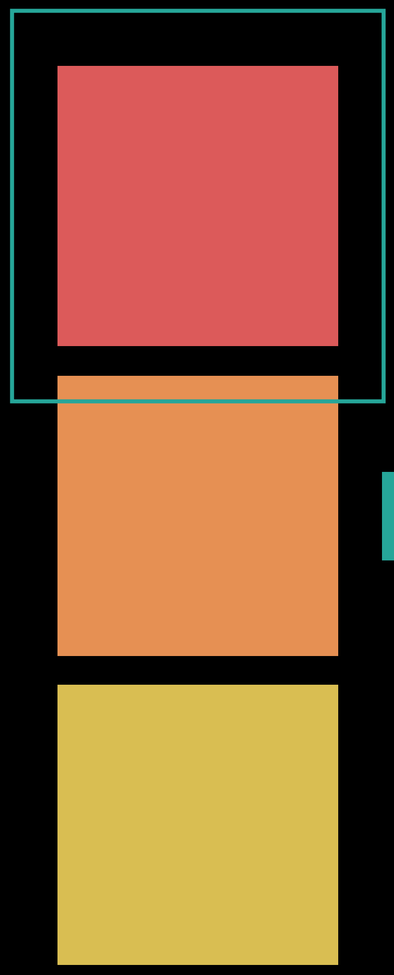

didAppear



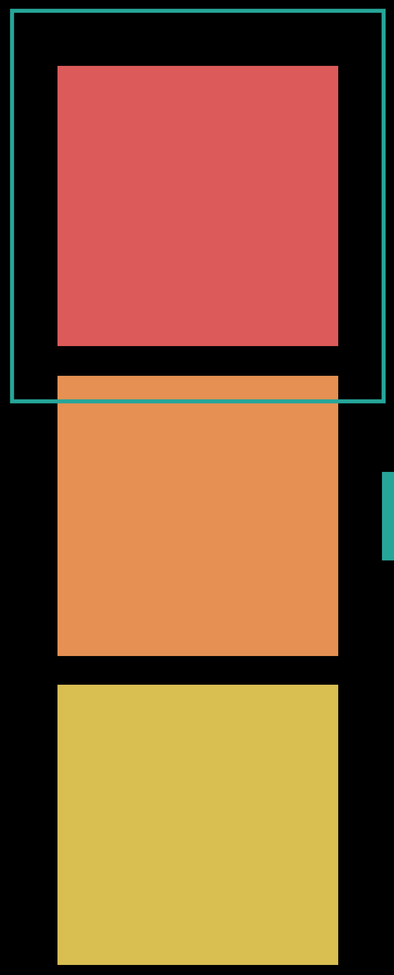
awakeWithContext



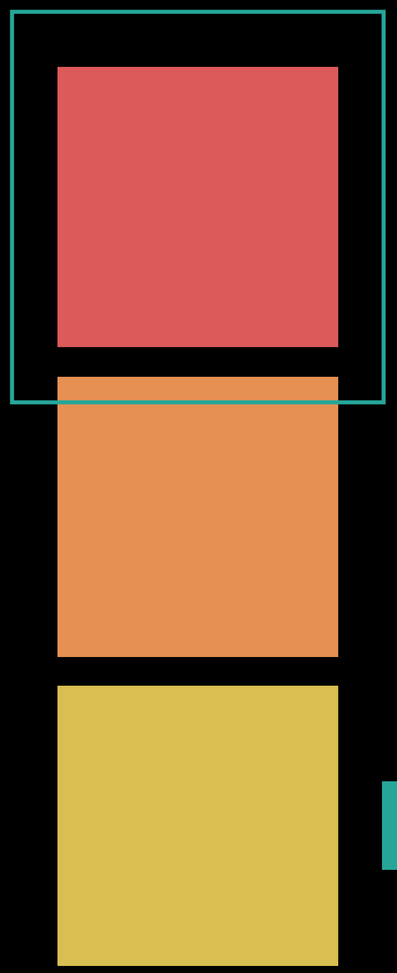
awakeWithContext



willActivate

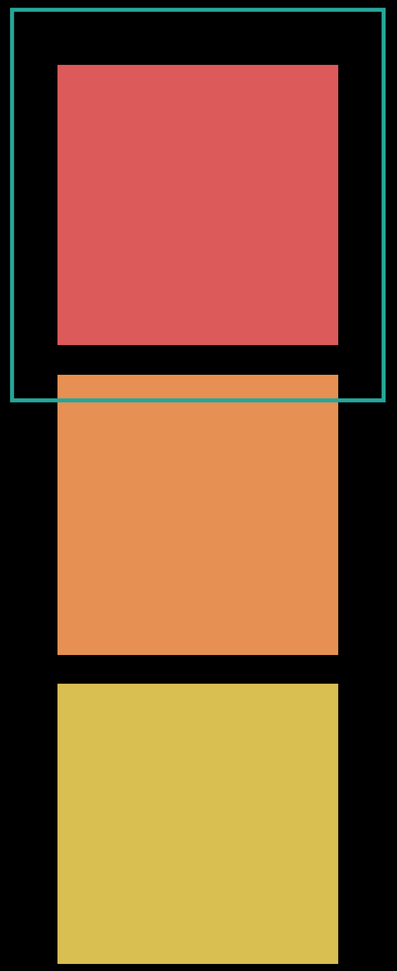


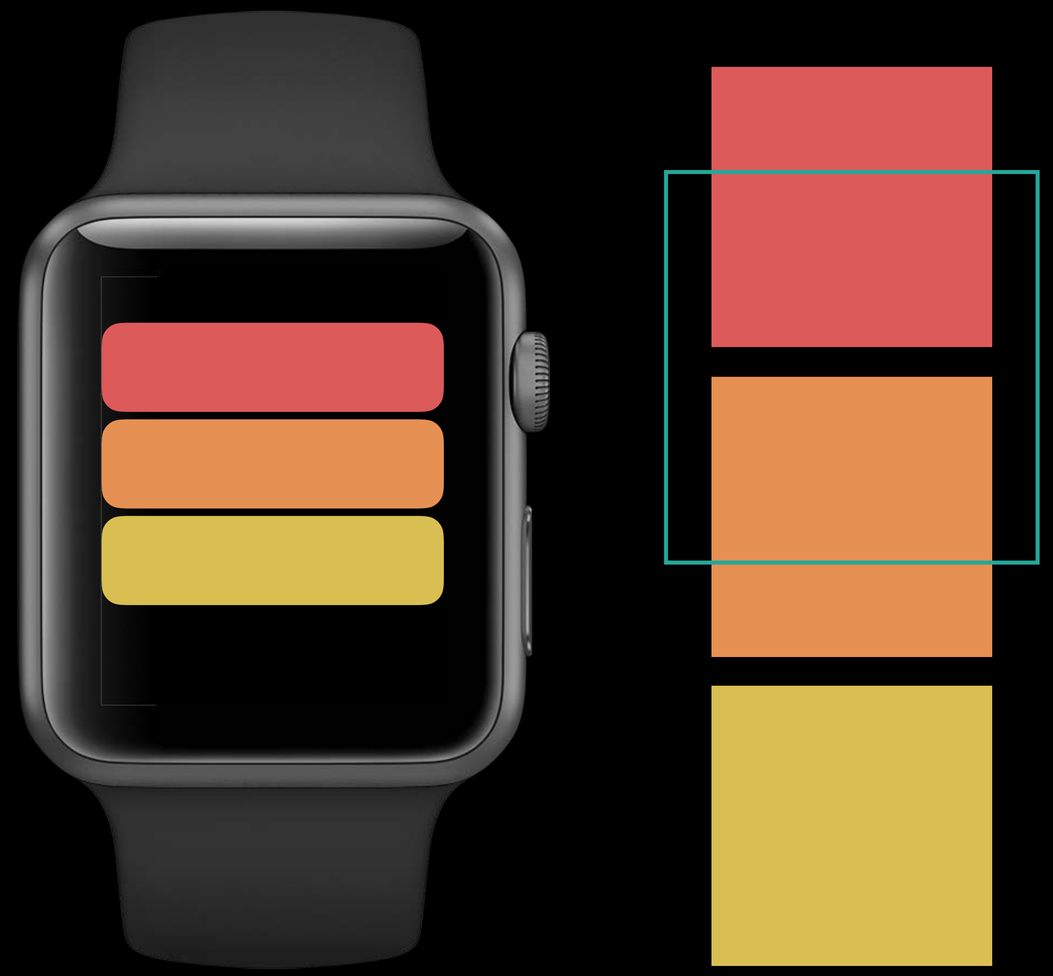
didDeactivate

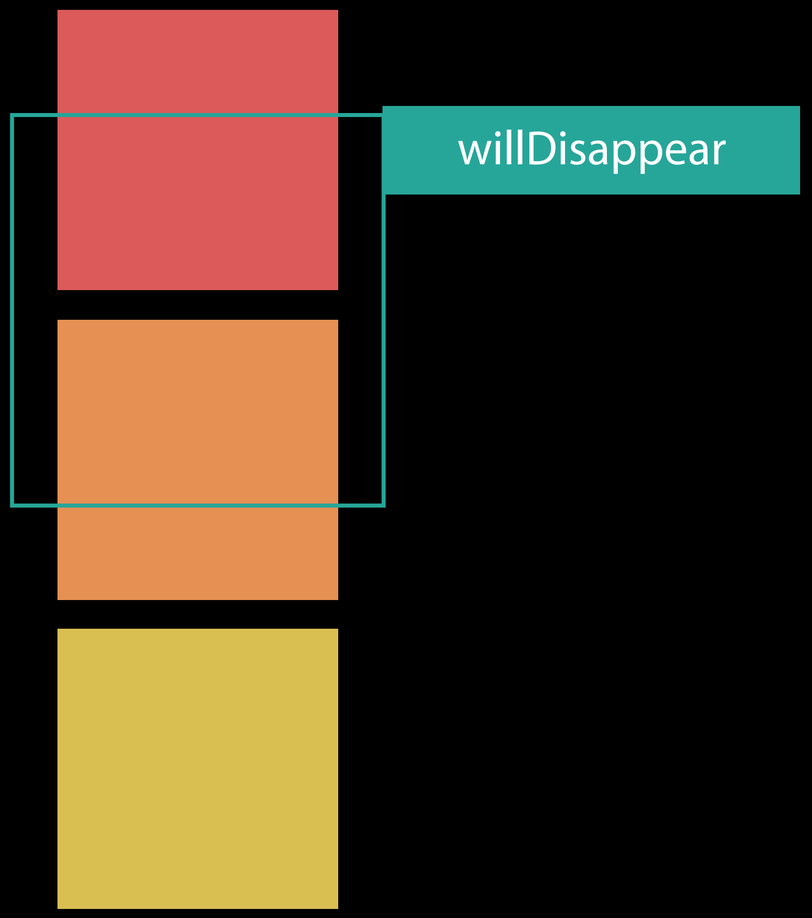


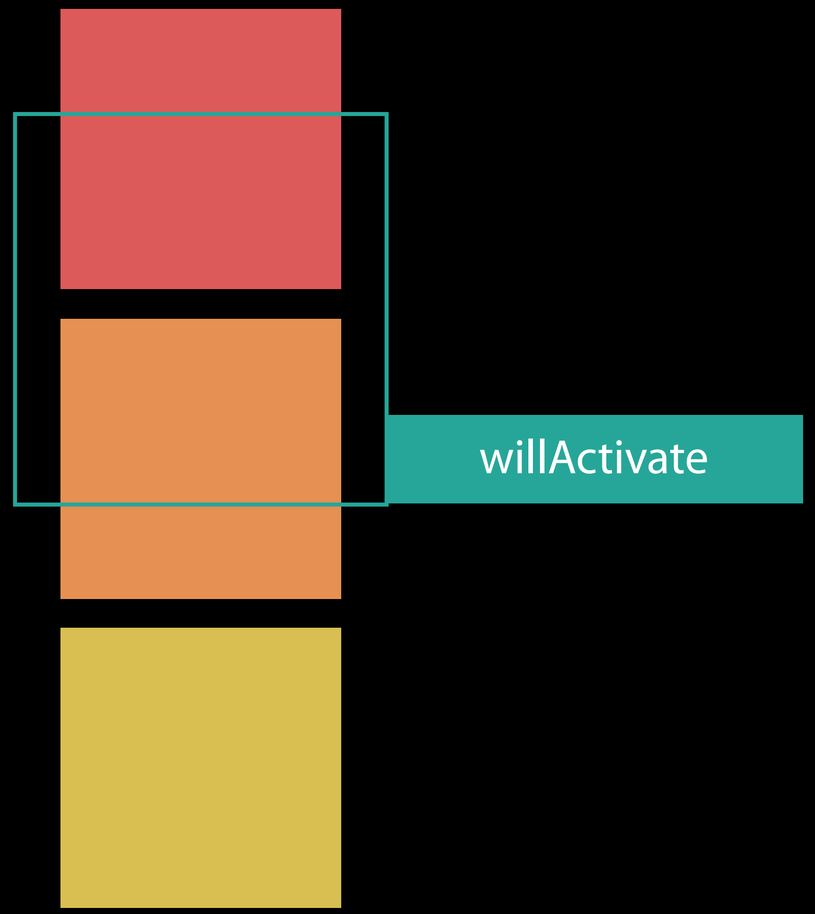
willActivate

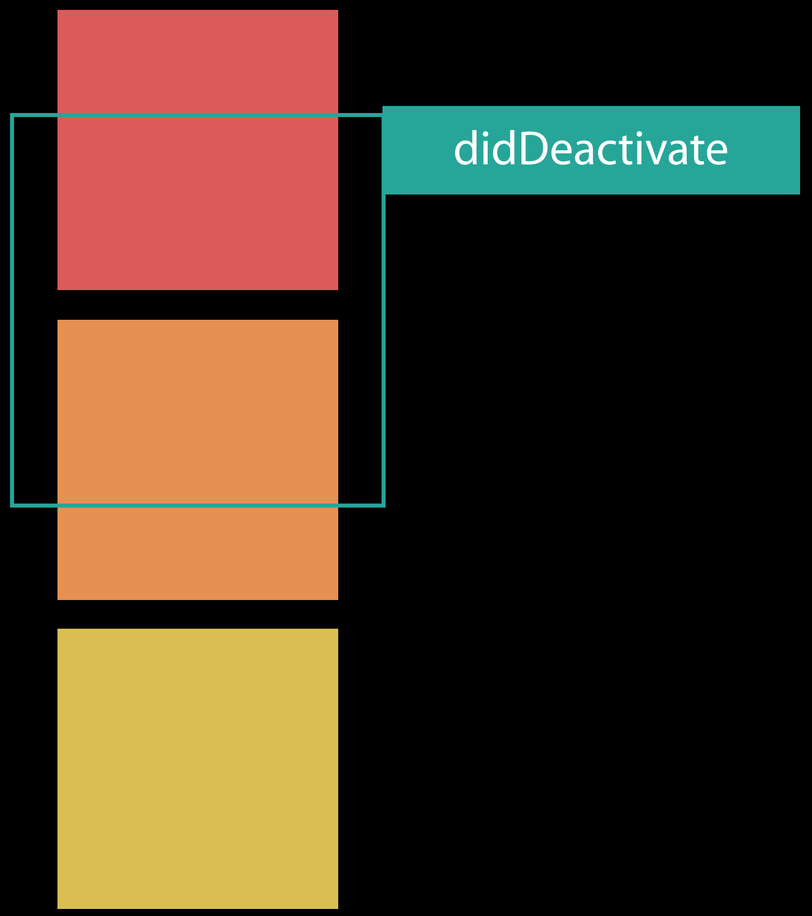
didDeactivate

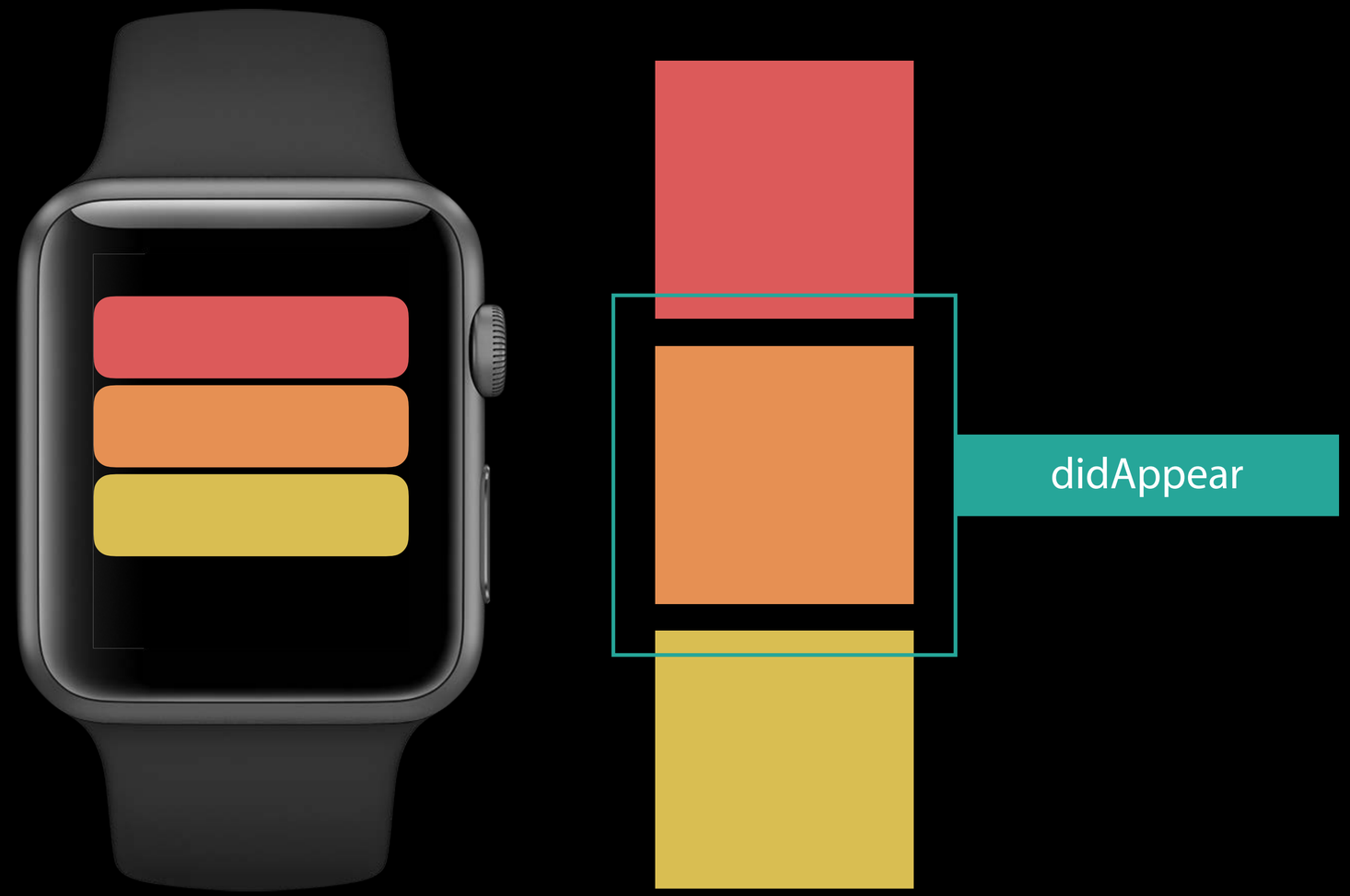


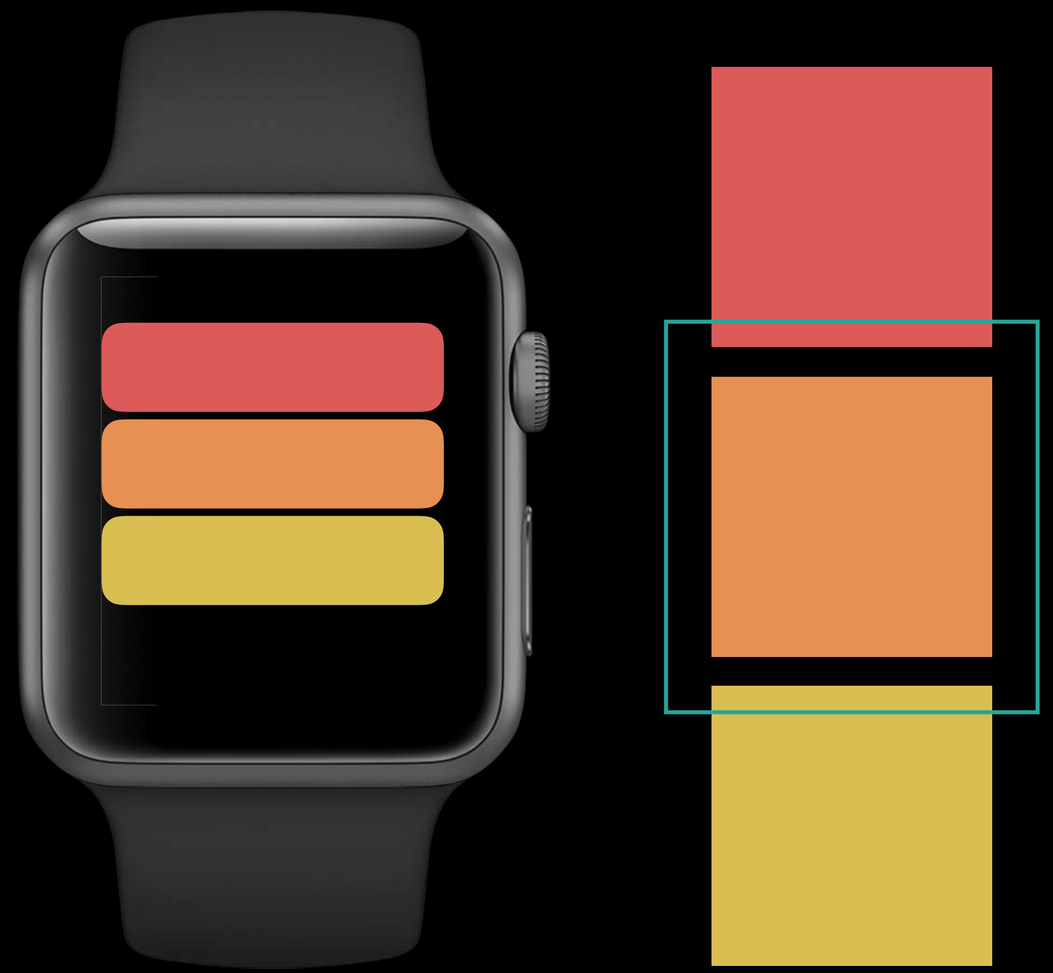


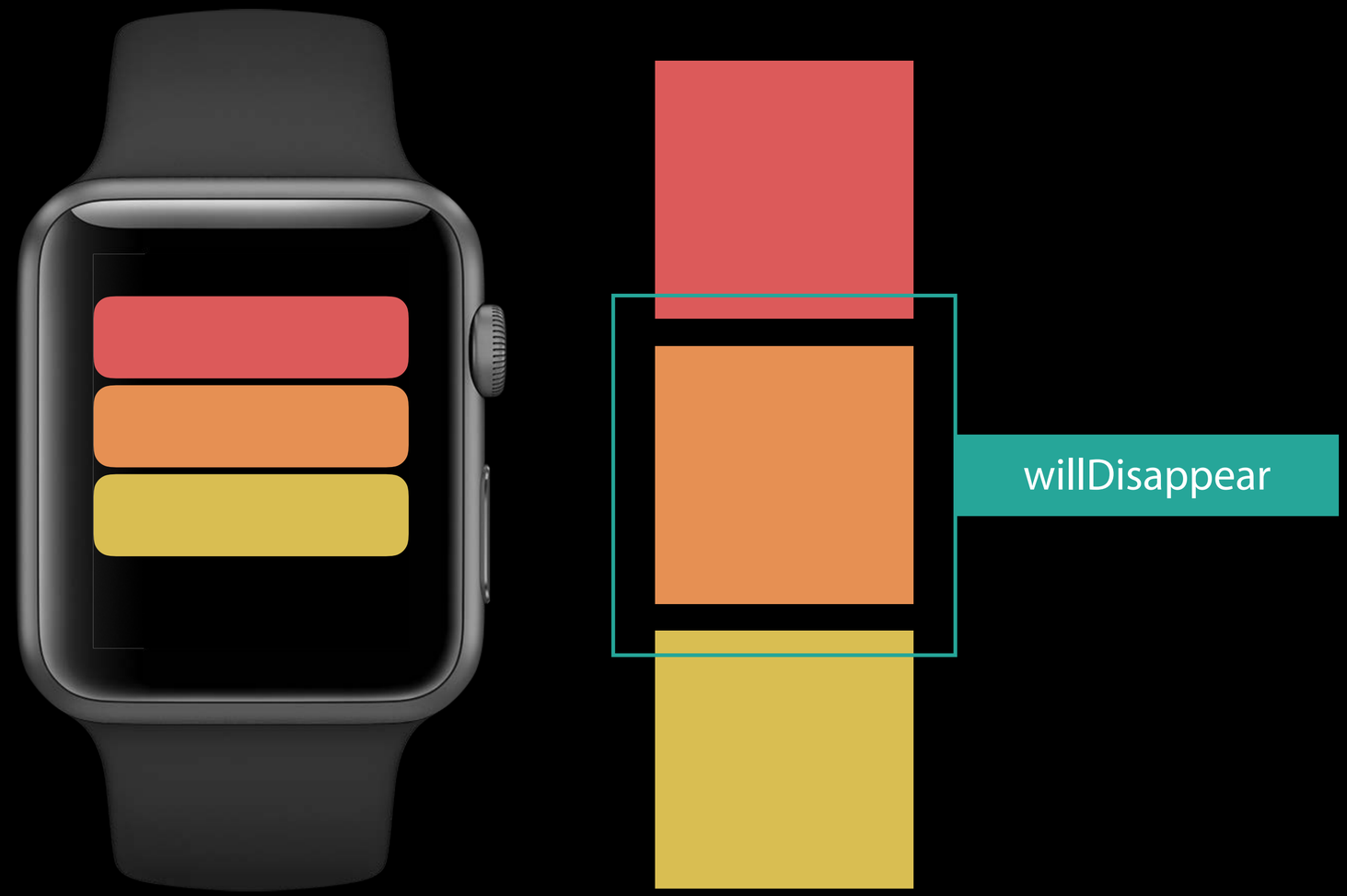

willDisappear

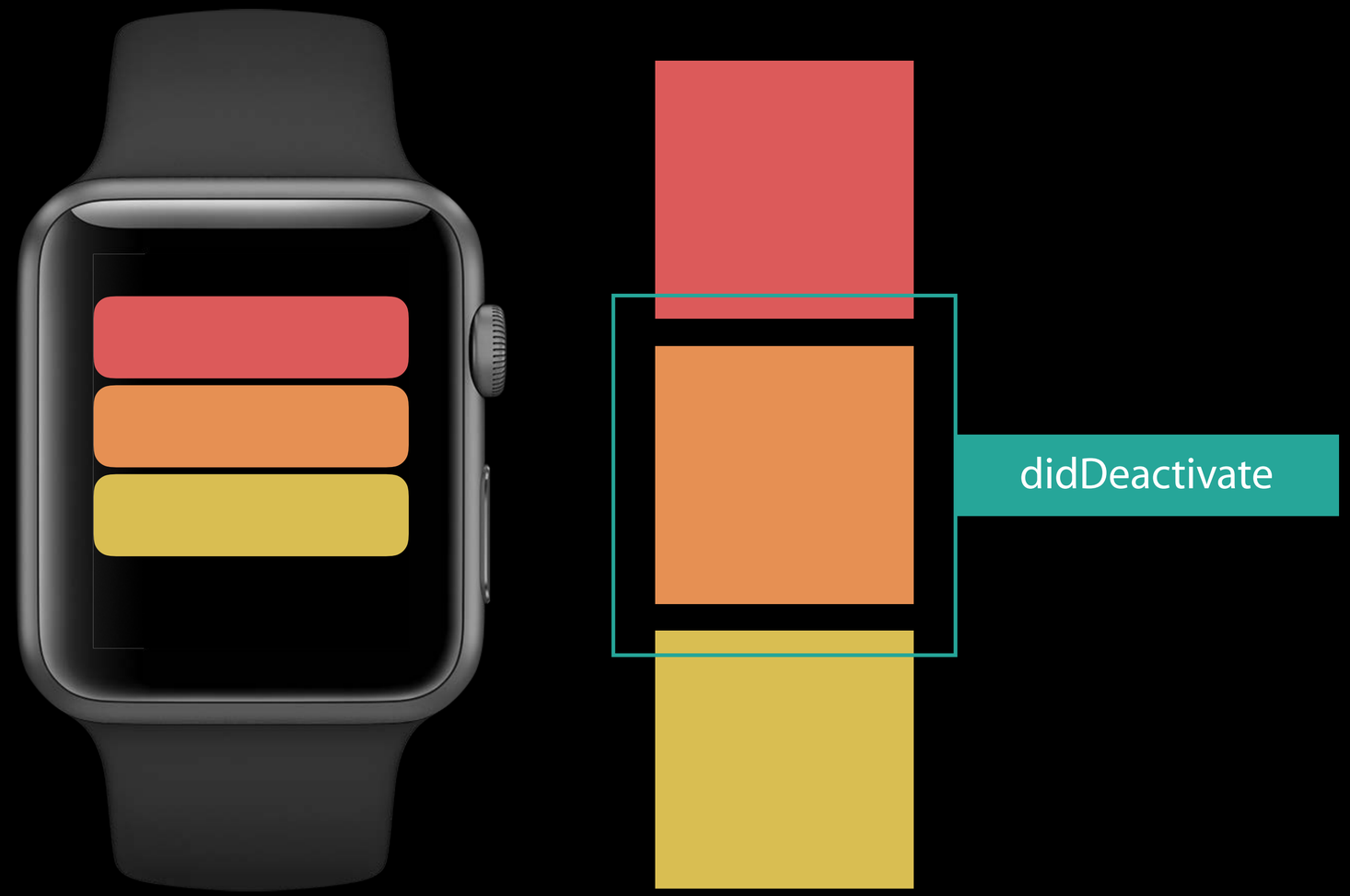

willActivate


didDeactivate


didAppear




willDisappear


didDeactivate

Case Study: Stocks

A Watch App built with WatchKit

Stocks

Case study outline

Stocks

Case study outline

2-second tasks

Stocks

Case study outline

2-second tasks

Background refresh

Stocks

Case study outline

2-second tasks

Background refresh

Resume time optimizations

2-Second Tasks

Stocks

2-second tasks

Complication

Chart/Detail

List View

2-Second Tasks

Complication

Fastest way to check your favorite stock's current price

Data is in sync between complication and app

2-Second Tasks

Complication

Fastest way to check your favorite stock's current price

Data is in sync between complication and app

2-Second Tasks

Navigation flow

watchOS 2

2-Second Tasks

Navigation flow

watchOS 2

2-Second Tasks

Navigation flow

watchOS 2

2-Second Tasks

Navigation flow

watchOS 2

2-Second Tasks

Navigation flow

watchOS 2

2-Second Tasks

Navigation flow

watchOS 2

watchOS 3

2-Second Tasks

Navigation flow

watchOS 2

watchOS 3

2-Second Tasks

In the Dock

2-Second Tasks

In the Dock

Stocks

2-second tasks recap

Stocks

2-second tasks recap

Consistent data between complication and app

Stocks

2-second tasks recap

Consistent data between complication and app

Simplified our design

Stocks

2-second tasks recap

Consistent data between complication and app

Simplified our design

Implemented new Vertical Detail Paging API

Background Refresh

Stocks

Background refresh

Stocks

Background refresh

How often do we need to update?

Stocks

Background refresh

How often do we need to update?

What data do we need to fetch to keep our app fresh?

Background Refresh

Refresh cadence

Background Refresh

Refresh cadence

Request data every 15 minutes

Background Refresh

Refresh cadence

Request data every 15 minutes

Markets are open for a period of time throughout the day

Background Refresh

Refresh cadence

Request data every 15 minutes

Markets are open for a period of time throughout the day

Background Refresh

Refresh cadence

Request data every 15 minutes

Markets are open for a period of time throughout the day

There are days when the market is not open

Background Refresh

Decide next refresh date

Background Refresh

Decide next refresh date

Enumerate through list of stocks

Background Refresh

Decide next refresh date

Enumerate through list of stocks

If markets are all closed, use earliest market open time

Background Refresh

Decide next refresh date

Enumerate through list of stocks

If markets are all closed, use earliest market open time

Else at least one market is open, use regular 15 minute cadence

```
// Schedule background refresh

func scheduleBackgroundRefresh(preferredDate: NSDate?) {
    if let preferredDate = preferredDate {
      let completion: (NSError?) -> Void = { error in
        // Handle error if needed
      }
      WKExtension.shared().scheduleBackgroundRefresh(
        withPreferredDate: preferredDate,
        userInfo: nil,
        scheduledCompletion: completion
      )
    }
}}
```

```
// Schedule background refresh

func scheduleBackgroundRefresh(preferredDate: NSDate?) {
    if let preferredDate = preferredDate {
      let completion: (NSError?) -> Void = { error in
        // Handle error if needed
      }
      WKExtension.shared().scheduleBackgroundRefresh(
        withPreferredDate: preferredDate,
        userInfo: nil,
        scheduledCompletion: completion
      )
    }
}}
```

```swift
// Schedule background refresh

func scheduleBackgroundRefresh(preferredDate: NSDate?) {
    if let preferredDate = preferredDate {
      let completion: (NSError?) -> Void = { error in
        // Handle error if needed
      }
      WKExtension.shared().scheduleBackgroundRefresh(
        withPreferredDate: preferredDate,
        userInfo: nil,
        scheduledCompletion: completion
      )
    }
}}
```

```
// Schedule background refresh

func scheduleBackgroundRefresh(preferredDate: NSDate?) {
    if let preferredDate = preferredDate {
      let completion: (NSError?) -> Void = { error in
        // Handle error if needed
      }
      WKExtension.shared().scheduleBackgroundRefresh(
        withPreferredDate: preferredDate,
        userInfo: nil,
        scheduledCompletion: completion
      )
    }
}}
```

```swift
// Helper method to grab preferred refresh date

func nextPreferredRefreshDate() -> NSDate? {
    guard let earliestNextOpenDateInStocks = self.earliestNextOpenDateInStocks() else {
      return nil
    }
    let nextRegularRefreshDate = NSDate(timeIntervalSinceNow: RefreshTimeInterval)
    return earliestNextOpenDateInStocks.laterDate(nextRegularRefreshDate)
}
```

```swift
// Helper method to grab preferred refresh date

func nextPreferredRefreshDate() -> NSDate? {
    guard let earliestNextOpenDateInStocks = self.earliestNextOpenDateInStocks() else {
      return nil
    }
    let nextRegularRefreshDate = NSDate(timeIntervalSinceNow: RefreshTimeInterval)
    return earliestNextOpenDateInStocks.laterDate(nextRegularRefreshDate)
}
```

```swift
// Helper method to grab preferred refresh date

func nextPreferredRefreshDate() -> NSDate? {
    guard let earliestNextOpenDateInStocks = self.earliestNextOpenDateInStocks() else {
      return nil
    }
    let nextRegularRefreshDate = NSDate(timeIntervalSinceNow: RefreshTimeInterval)
    return earliestNextOpenDateInStocks.laterDate(nextRegularRefreshDate)
}
```

```swift
// Helper method to grab preferred refresh date

func nextPreferredRefreshDate() -> NSDate? {
    guard let earliestNextOpenDateInStocks = self.earliestNextOpenDateInStocks() else {
      return nil
    }
    let nextRegularRefreshDate = NSDate(timeIntervalSinceNow: RefreshTimeInterval)
    return earliestNextOpenDateInStocks.laterDate(nextRegularRefreshDate)
}
```

```swift
// Helper method to grab preferred refresh date

func nextPreferredRefreshDate() -> NSDate? {
    guard let earliestNextOpenDateInStocks = self.earliestNextOpenDateInStocks() else {
      return nil
    }
    let nextRegularRefreshDate = NSDate(timeIntervalSinceNow: RefreshTimeInterval)
    return earliestNextOpenDateInStocks.laterDate(nextRegularRefreshDate)
}
```

```swift
// Calculate the “next open date” for a user’s list of stocks

func earliestNextOpenDateInStocks() -> NSDate? {
    let stocks = self.stocksManager.stocks
    guard stocks.count > 0 else {
      return nil
    }
    var earliestNextOpenDate = NSDate.distantFuture()
    for stock in stocks {
      guard !stock.marketIsOpen else {
        // If market is open, return distantPast
        return NSDate.distantPast()
      }
      if let nextMarketOpenDate = stock.nextMarketOpenDate {
        earliestNextOpenDate = nextMarketOpenDate.earlierDate(earliestNextOpenDate)
      }
    }
    return earliestNextOpenDate
}
```

```swift
// Calculate the “next open date” for a user’s list of stocks

func earliestNextOpenDateInStocks() -> NSDate? {
    let stocks = self.stocksManager.stocks
    guard stocks.count > 0 else {
      return nil
    }
    var earliestNextOpenDate = NSDate.distantFuture()
    for stock in stocks {
      guard !stock.marketIsOpen else {
        // If market is open, return distantPast
        return NSDate.distantPast()
      }
      if let nextMarketOpenDate = stock.nextMarketOpenDate {
        earliestNextOpenDate = nextMarketOpenDate.earlierDate(earliestNextOpenDate)
      }
    }
    return earliestNextOpenDate
}
```

```swift
// Calculate the “next open date” for a user’s list of stocks

func earliestNextOpenDateInStocks() -> NSDate? {
    let stocks = self.stocksManager.stocks
    guard stocks.count > 0 else {
      return nil
    }
    var earliestNextOpenDate = NSDate.distantFuture()
    for stock in stocks {
      guard !stock.marketIsOpen else {
        // If market is open, return distantPast
        return NSDate.distantPast()
      }
      if let nextMarketOpenDate = stock.nextMarketOpenDate {
        earliestNextOpenDate = nextMarketOpenDate.earlierDate(earliestNextOpenDate)
      }
    }
    return earliestNextOpenDate
}
```

```swift
// Calculate the “next open date” for a user’s list of stocks

func earliestNextOpenDateInStocks() -> NSDate? {
    let stocks = self.stocksManager.stocks
    guard stocks.count > 0 else {
      return nil
    }
    var earliestNextOpenDate = NSDate.distantFuture()
    for stock in stocks {
      guard !stock.marketIsOpen else {
        // If market is open, return distantPast
        return NSDate.distantPast()
      }
      if let nextMarketOpenDate = stock.nextMarketOpenDate {
        earliestNextOpenDate = nextMarketOpenDate.earlierDate(earliestNextOpenDate)
      }
    }
    return earliestNextOpenDate
}
```

```swift
// Calculate the “next open date” for a user’s list of stocks

func earliestNextOpenDateInStocks() -> NSDate? {
    let stocks = self.stocksManager.stocks
    guard stocks.count > 0 else {
      return nil
    }
    var earliestNextOpenDate = NSDate.distantFuture()
    for stock in stocks {
      guard !stock.marketIsOpen else {
        // If market is open, return distantPast
        return NSDate.distantPast()
      }
      if let nextMarketOpenDate = stock.nextMarketOpenDate {
        earliestNextOpenDate = nextMarketOpenDate.earlierDate(earliestNextOpenDate)
      }
    }
    return earliestNextOpenDate
}
```

```
// Calculate the “next open date” for a user’s list of stocks

func earliestNextOpenDateInStocks() -> NSDate? {
    let stocks = self.stocksManager.stocks
    guard stocks.count > 0 else {
      return nil
    }
    var earliestNextOpenDate = NSDate.distantFuture()
    for stock in stocks {
      guard !stock.marketIsOpen else {
        // If market is open, return distantPast
        return NSDate.distantPast()
      }
      if let nextMarketOpenDate = stock.nextMarketOpenDate {
        earliestNextOpenDate = nextMarketOpenDate.earlierDate(earliestNextOpenDate)
      }
    }
    return earliestNextOpenDate
}
```

```swift
// Calculate the “next open date” for a user’s list of stocks

func earliestNextOpenDateInStocks() -> NSDate? {
    let stocks = self.stocksManager.stocks
    guard stocks.count > 0 else {
      return nil
    }
    var earliestNextOpenDate = NSDate.distantFuture()
    for stock in stocks {
      guard !stock.marketIsOpen else {
        // If market is open, return distantPast
        return NSDate.distantPast()
      }
      if let nextMarketOpenDate = stock.nextMarketOpenDate {
        earliestNextOpenDate = nextMarketOpenDate.earlierDate(earliestNextOpenDate)
      }
    }
    return earliestNextOpenDate
}
```

Background Refresh

Schedule multiple background requests

Background Refresh

Schedule multiple background requests

Endpoint A updates the app

Endpoint B updates the complication

Background Refresh

Schedule multiple background requests

Schedule background refresh time

On receipt

- Submit Endpoint A request
- Submit Endpoint B request
- Schedule future background refresh time

```swift
// WKExtensionDelegate Handle Background Tasks

func handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>) {
    for task in backgroundTasks {
      switch task {
      case let appRefreshTask as WKApplicationRefreshBackgroundTask:
        self.scheduleDataUpdateRequest()
        self.scheduleBackgroundRefresh(preferredDate: self.nextPreferredRefreshDate())
        appRefreshTask.setTaskCompleted()
      case let urlSessionTask as WKURLSessionRefreshBackgroundTask:
        self.storeURLSessionTask(urlSessionTask: urlSessionTask)
      default:
        task.setTaskCompleted()
      }
    }
}
```

```swift
// WKExtensionDelegate Handle Background Tasks

func handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>) {
    for task in backgroundTasks {
      switch task {
      case let appRefreshTask as WKApplicationRefreshBackgroundTask:
        self.scheduleDataUpdateRequest()
        self.scheduleBackgroundRefresh(preferredDate: self.nextPreferredRefreshDate())
        appRefreshTask.setTaskCompleted()
      case let urlSessionTask as WKURLSessionRefreshBackgroundTask:
        self.storeURLSessionTask(urlSessionTask: urlSessionTask)
      default:
        task.setTaskCompleted()
      }
    }
}
```

```swift
// WKExtensionDelegate Handle Background Tasks

func handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>) {
    for task in backgroundTasks {
      switch task {
      case let appRefreshTask as WKApplicationRefreshBackgroundTask:
        self.scheduleDataUpdateRequest()
        self.scheduleBackgroundRefresh(preferredDate: self.nextPreferredRefreshDate())
        appRefreshTask.setTaskCompleted()
      case let urlSessionTask as WKURLSessionRefreshBackgroundTask:
        self.storeURLSessionTask(urlSessionTask: urlSessionTask)
      default:
        task.setTaskCompleted()
      }
    }
}
```

```swift
// WKExtensionDelegate Handle Background Tasks

func handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>) {
    for task in backgroundTasks {
      switch task {
      case let appRefreshTask as WKApplicationRefreshBackgroundTask:
        self.scheduleDataUpdateRequest()
        self.scheduleBackgroundRefresh(preferredDate: self.nextPreferredRefreshDate())
        appRefreshTask.setTaskCompleted()
      case let urlSessionTask as WKURLSessionRefreshBackgroundTask:
        self.storeURLSessionTask(urlSessionTask: urlSessionTask)
      default:
        task.setTaskCompleted()
      }
    }
}
```

```swift
// WKExtensionDelegate Handle Background Tasks

func handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>) {
    for task in backgroundTasks {
      switch task {
      case let appRefreshTask as WKApplicationRefreshBackgroundTask:
        self.scheduleDataUpdateRequest()
        self.scheduleBackgroundRefresh(preferredDate: self.nextPreferredRefreshDate())
        appRefreshTask.setTaskCompleted()
      case let urlSessionTask as WKURLSessionRefreshBackgroundTask:
        self.storeURLSessionTask(urlSessionTask: urlSessionTask)
      default:
        task.setTaskCompleted()
      }
    }
}
```

```swift
// WKExtensionDelegate Handle Background Tasks

func handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>) {
    for task in backgroundTasks {
      switch task {
      case let appRefreshTask as WKApplicationRefreshBackgroundTask:
        self.scheduleDataUpdateRequest()
        self.scheduleBackgroundRefresh(preferredDate: self.nextPreferredRefreshDate())
        appRefreshTask.setTaskCompleted()
      case let urlSessionTask as WKURLSessionRefreshBackgroundTask:
        self.storeURLSessionTask(urlSessionTask: urlSessionTask)
      default:
        task.setTaskCompleted()
      }
    }
}
```

```swift
// WKExtensionDelegate Handle Background Tasks

func handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>) {
    for task in backgroundTasks {
      switch task {
      case let appRefreshTask as WKApplicationRefreshBackgroundTask:
        self.scheduleDataUpdateRequest()
        self.scheduleBackgroundRefresh(preferredDate: self.nextPreferredRefreshDate())
        appRefreshTask.setTaskCompleted()
      case let urlSessionTask as WKURLSessionRefreshBackgroundTask:
        self.storeURLSessionTask(urlSessionTask: urlSessionTask)
      default:
        task.setTaskCompleted()
      }
    }
}
```

```swift
// WKExtensionDelegate Handle Background Tasks

func handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>) {
    for task in backgroundTasks {
      switch task {
      case let appRefreshTask as WKApplicationRefreshBackgroundTask:
        self.scheduleDataUpdateRequest()
        self.scheduleBackgroundRefresh(preferredDate: self.nextPreferredRefreshDate())
        appRefreshTask.setTaskCompleted()
      case let urlSessionTask as WKURLSessionRefreshBackgroundTask:
        self.storeURLSessionTask(urlSessionTask: urlSessionTask)
      default:
        task.setTaskCompleted()
      }
    }
}
```

```swift
// WKExtensionDelegate Handle Background Tasks

func handle(_ backgroundTasks: Set<WKRefreshBackgroundTask>) {
    for task in backgroundTasks {
      switch task {
      case let appRefreshTask as WKApplicationRefreshBackgroundTask:
        self.scheduleDataUpdateRequest()
        self.scheduleBackgroundRefresh(preferredDate: self.nextPreferredRefreshDate())
        appRefreshTask.setTaskCompleted()
      case let urlSessionTask as WKURLSessionRefreshBackgroundTask:
        self.storeURLSessionTask(urlSessionTask: urlSessionTask)
      default:
        task.setTaskCompleted()
      }
    }
}
```

Background Refresh

Schedule app update background request

Schedule requests

On complete of requests

- Complete `WKURLSessionRefreshBackgroundTask`
- Schedule snapshot
- Reload complication

```swift
// Schedule the network request to keep the app snapshot up-to-date

func scheduleDataUpdateRequest() {
    // Setup download tasks
    self.setupAppDataRequest()
    self.setupComplicationDataRequest()
    // Setup finishUpdateHandler
    self.finishUpdateHandler = { sessionIdentifier -> Void in
      if let taskToComplete = self.urlSessionTasks[sessionIdentifier] {
        self.scheduleSnapshot()
        self.reloadComplication()
        taskToComplete.setTaskCompleted()
      }
    }
    self.submitRequests()
}
```

```swift
// Schedule the network request to keep the app snapshot up-to-date

func scheduleDataUpdateRequest() {
    // Setup download tasks
    self.setupAppDataRequest()
    self.setupComplicationDataRequest()
    // Setup finishUpdateHandler
    self.finishUpdateHandler = { sessionIdentifier -> Void in
      if let taskToComplete = self.urlSessionTasks[sessionIdentifier] {
        self.scheduleSnapshot()
        self.reloadComplication()
        taskToComplete.setTaskCompleted()
      }
    }
    self.submitRequests()
}
```

```
// Schedule the network request to keep the app snapshot up-to-date

func scheduleDataUpdateRequest() {
    // Setup download tasks
    self.setupAppDataRequest()
    self.setupComplicationDataRequest()
    // Setup finishUpdateHandler
    self.finishUpdateHandler = { sessionIdentifier -> Void in
      if let taskToComplete = self.urlSessionTasks[sessionIdentifier] {
        self.scheduleSnapshot()
        self.reloadComplication()
        taskToComplete.setTaskCompleted()
      }
    }
    self.submitRequests()
}
```

```
// Schedule the network request to keep the app snapshot up-to-date

func scheduleDataUpdateRequest() {
    // Setup download tasks
    self.setupAppDataRequest()
    self.setupComplicationDataRequest()
    // Setup finishUpdateHandler
    self.finishUpdateHandler = { sessionIdentifier -> Void in
      if let taskToComplete = self.urlSessionTasks[sessionIdentifier] {
        self.scheduleSnapshot()
        self.reloadComplication()
        taskToComplete.setTaskCompleted()
      }
    }
    self.submitRequests()
}
```

```swift
// Schedule the network request to keep the app snapshot up-to-date

func scheduleDataUpdateRequest() {
    // Setup download tasks
    self.setupAppDataRequest()
    self.setupComplicationDataRequest()
    // Setup finishUpdateHandler
    self.finishUpdateHandler = { sessionIdentifier -> Void in
      if let taskToComplete = self.urlSessionTasks[sessionIdentifier] {
        self.scheduleSnapshot()
        self.reloadComplication()
        taskToComplete.setTaskCompleted()
      }
    }
    self.submitRequests()
}
```

```swift
// Schedule the network request to keep the app snapshot up-to-date

func scheduleDataUpdateRequest() {
    // Setup download tasks
    self.setupAppDataRequest()
    self.setupComplicationDataRequest()
    // Setup finishUpdateHandler
    self.finishUpdateHandler = { sessionIdentifier -> Void in
      if let taskToComplete = self.urlSessionTasks[sessionIdentifier] {
        self.scheduleSnapshot()
        self.reloadComplication()
        taskToComplete.setTaskCompleted()
      }
    }
    self.submitRequests()
}
```

```swift
// NSURLSessionDelegate Background Download Tasks Complete

@objc func urlSessionDidFinishEvents(forBackgroundURLSession session: NSURLSession) {
    if let identifier = session.configuration.identifier,
      finishUpdateHandler = self.finishUpdateHandler {
      finishUpdateHandler(identifier)
    }
}
```

```swift
// NSURLSessionDelegate Background Download Tasks Complete

@objc func urlSessionDidFinishEvents(forBackgroundURLSession session: NSURLSession) {
    if let identifier = session.configuration.identifier,
      finishUpdateHandler = self.finishUpdateHandler {
      finishUpdateHandler(identifier)
    }
}
```

```swift
// NSURLSessionDelegate Background Download Tasks Complete

@objc func urlSessionDidFinishEvents(forBackgroundURLSession session: NSURLSession) {
    if let identifier = session.configuration.identifier,
      finishUpdateHandler = self.finishUpdateHandler {
      finishUpdateHandler(identifier)
    }
}
```

Stocks

Background refresh recap

Optimize how often you schedule updates for your app

If updating with data from a server, try to use single specialized endpoint

Resume Time Optimizations

Stocks

Resume time

Minimize work during `willActivate` and `didAppear`

- Avoid long running tasks that are triggered from `willActivate`
- Smart load/reload of data
- Only set properties on WKInterfaceObjects that have changed

Resume Time

Cautionary tale for Vertical Detail Paging API

Neighboring detail pages will have `willActivate` called

Avoid expensive operations in `willActivate` for detail pages

Resume Time

Cautionary tale for Vertical Detail Paging API

Neighboring detail pages will have `willActivate` called

Avoid expensive operations in `willActivate` for detail pages

Resume Time

Cautionary tale for Vertical Detail Paging API

Reports of slow loading chart when entering first detail page

Other detail pages never finished loading their charts

```swift
// Initial Approach - willActivate/didAppear in StockInterfaceController.swift

override func willActivate() {
    super.willActivate()
    downloadAndGenerateChart()
}
override func didAppear() {
    super.didAppear()
}
func downloadAndGenerateChart() {
    // Long running task to download chart data and generate chart image
}
```

```swift
// Initial Approach - willActivate/didAppear in StockInterfaceController.swift

override func willActivate() {
    super.willActivate()
    downloadAndGenerateChart()
}
override func didAppear() {
    super.didAppear()
}
func downloadAndGenerateChart() {
    // Long running task to download chart data and generate chart image
}
```

```swift
// Initial Approach - willActivate/didAppear in StockInterfaceController.swift

override func willActivate() {
    super.willActivate()
    downloadAndGenerateChart()
}
override func didAppear() {
    super.didAppear()
}
func downloadAndGenerateChart() {
    // Long running task to download chart data and generate chart image
}
```

```swift
// Better Approach - willActivate/didAppear in StockInterfaceController.swift

override func willActivate() {
    super.willActivate()
}
override func didAppear() {
    super.didAppear()
    downloadAndGenerateChart()
}
override func willDisappear() {
    super.willDisappear()
    cancelDownloadAndGenerateChart()
}
func downloadAndGenerateChart() {
    // Long running task to download chart data and generate chart image
}
func cancelDownloadAndGenerateChart() {
    //Cancel long running task to download chart data and generate chart image
}
```

```swift
// Better Approach - willActivate/didAppear in StockInterfaceController.swift

override func willActivate() {
    super.willActivate()
}
override func didAppear() {
    super.didAppear()
    downloadAndGenerateChart()
}
override func willDisappear() {
    super.willDisappear()
    cancelDownloadAndGenerateChart()
}
func downloadAndGenerateChart() {
    // Long running task to download chart data and generate chart image
}
func cancelDownloadAndGenerateChart() {
    //Cancel long running task to download chart data and generate chart image
}
```

```swift
// Better Approach - willActivate/didAppear in StockInterfaceController.swift

override func willActivate() {
    super.willActivate()
}
override func didAppear() {
    super.didAppear()
    downloadAndGenerateChart()
}
override func willDisappear() {
    super.willDisappear()
    cancelDownloadAndGenerateChart()
}
func downloadAndGenerateChart() {
    // Long running task to download chart data and generate chart image
}
func cancelDownloadAndGenerateChart() {
    //Cancel long running task to download chart data and generate chart image
}
```

```swift
// Better Approach - willActivate/didAppear in StockInterfaceController.swift

override func willActivate() {
    super.willActivate()
}
override func didAppear() {
    super.didAppear()
    downloadAndGenerateChart()
}
override func willDisappear() {
    super.willDisappear()
    cancelDownloadAndGenerateChart()
}
func downloadAndGenerateChart() {
    // Long running task to download chart data and generate chart image
}
func cancelDownloadAndGenerateChart() {
    //Cancel long running task to download chart data and generate chart image
}
```

Resume Time

Vertical Detail Paging API caveats

Resume Time

Vertical Detail Paging API caveats

Avoid triggering long running tasks in `willActivate`

Resume Time

Vertical Detail Paging API caveats

Avoid triggering long running tasks in `willActivate`

Make use of cancelable operations

Resume Time

WKInterfaceTable loading

Resume Time

WKInterfaceTable loading

All rows are loaded in memory

Resume Time

WKInterfaceTable loading

All rows are loaded in memory

There is a linear upfront cost to the number of rows you have in your table

Resume Time

WKInterfaceTable loading

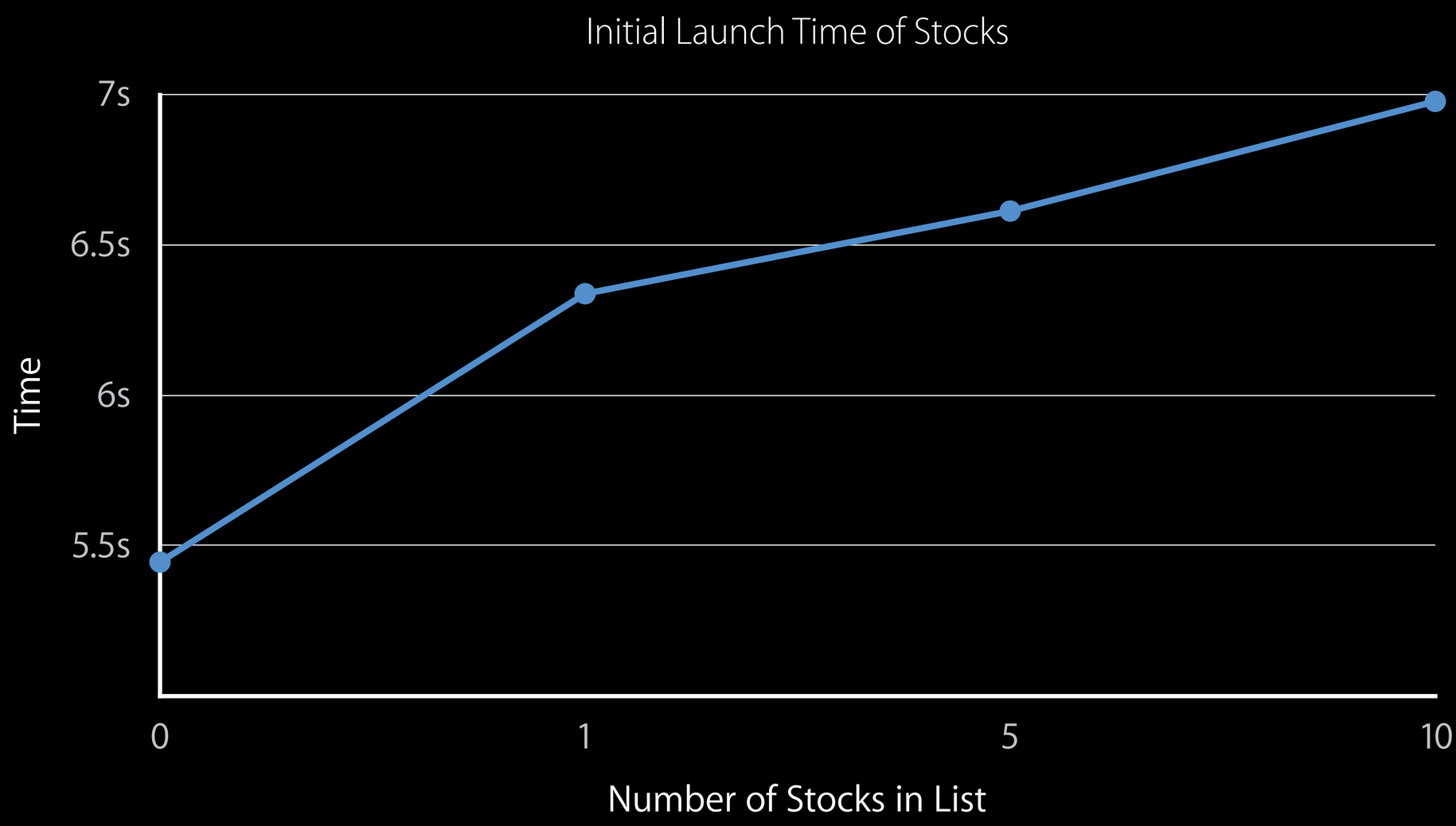
All rows are loaded in memory

There is a linear upfront cost to the number of rows you have in your table

No reuse

Resume Time

WKInterfaceTable loading



Resume Time

Improve WKInterfaceTable loading performance

Resume Time

Improve WKInterfaceTable loading performance

Limit the number of rows you load

Resume Time

Improve WKInterfaceTable loading performance

Limit the number of rows you load

Do smart updates of your table when row deltas occur

```
// Initial Approach - Load Stocks Table

func loadTable() {
    let stocks = self.stocksManager.stocks
    self.table.setNumberOfRows(stocks.count, withRowType: stockRowControllerIdentifier)
    for (index, stock) in stocks.enumerated() {
      self.populateStockRowController(index: index, stock: stock)
    }
}
```

```
// Initial Approach - Load Stocks Table

func loadTable() {
    let stocks = self.stocksManager.stocks
    self.table.setNumberOfRows(stocks.count, withRowType: stockRowControllerIdentifier)
    for (index, stock) in stocks.enumerated() {
      self.populateStockRowController(index: index, stock: stock)
    }
}
```

```
// Initial Approach - Load Stocks Table

func loadTable() {
    let stocks = self.stocksManager.stocks
    self.table.setNumberOfRows(stocks.count, withRowType: stockRowControllerIdentifier)
    for (index, stock) in stocks.enumerated() {
      self.populateStockRowController(index: index, stock: stock)
    }
}
```

Resume Time

Improve WKInterfaceTable loading performance

Resume Time

Improve WKInterfaceTable loading performance

Number of stocks in list is not capped

Resume Time

Improve WKInterfaceTable loading performance

Number of stocks in list is not capped

Always using

```
setNumberOfRows(numberOfRows: Int, withRowType rowType: String)
```

```swift
// Second Approach - Load Stocks Table

func loadTableSmart() {
    let stocks = self.stocksManager.stocks
    let stocksCount = min(stocks.count, maxStocksListSize)
    let stockRowDelta = stocksCount - self.table.numberOfRows
    self.insertRemoveTableRows(stockRowDelta: stockRowDelta)
    for ( index, stock ) in stocks.enumerated() {
      guard index < maxStocksListSize else {
        break
      }
      self.populateStockRowController(index: index, stock: stock)
    }
}
```

```swift
// Second Approach - Load Stocks Table

func loadTableSmart() {
    let stocks = self.stocksManager.stocks
    let stocksCount = min(stocks.count, maxStocksListSize)
    let stockRowDelta = stocksCount - self.table.numberOfRows
    self.insertRemoveTableRows(stockRowDelta: stockRowDelta)
    for ( index, stock ) in stocks.enumerated() {
      guard index < maxStocksListSize else {
        break
      }
      self.populateStockRowController(index: index, stock: stock)
    }
}
```

```swift
// Second Approach - Load Stocks Table

func loadTableSmart() {
    let stocks = self.stocksManager.stocks
    let stocksCount = min(stocks.count, maxStocksListSize)
    let stockRowDelta = stocksCount - self.table.numberOfRows
    self.insertRemoveTableRows(stockRowDelta: stockRowDelta)
    for ( index, stock ) in stocks.enumerated() {
      guard index < maxStocksListSize else {
        break
      }
      self.populateStockRowController(index: index, stock: stock)
    }
}
```

```
// Second Approach - Load Stocks Table

func loadTableSmart() {
    let stocks = self.stocksManager.stocks
    let stocksCount = min(stocks.count, maxStocksListSize)
    let stockRowDelta = stocksCount - self.table.numberOfRows
    self.insertRemoveTableRows(stockRowDelta: stockRowDelta)
    for ( index, stock ) in stocks.enumerated() {
      guard index < maxStocksListSize else {
        break
      }
      self.populateStockRowController(index: index, stock: stock)
    }
}
```

```swift
// Second Approach - Load Stocks Table

func loadTableSmart() {
    let stocks = self.stocksManager.stocks
    let stocksCount = min(stocks.count, maxStocksListSize)
    let stockRowDelta = stocksCount - self.table.numberOfRows
    self.insertRemoveTableRows(stockRowDelta: stockRowDelta)
    for ( index, stock ) in stocks.enumerated() {
      guard index < maxStocksListSize else {
        break
      }
      self.populateStockRowController(index: index, stock: stock)
    }
}
```

```swift
// Second Approach - Load Stocks Table

func loadTableSmart() {
    let stocks = self.stocksManager.stocks
    let stocksCount = min(stocks.count, maxStocksListSize)
    let stockRowDelta = stocksCount - self.table.numberOfRows
    self.insertRemoveTableRows(stockRowDelta: stockRowDelta)
    for ( index, stock ) in stocks.enumerated() {
      guard index < maxStocksListSize else {
        break
      }
      self.populateStockRowController(index: index, stock: stock)
    }
}
```

```
// Second Approach - Load Stocks Table - insert/remove table rows

func insertRemoveTableRows(stockRowDelta: Int) {
    let stockRowChangeRange = NSRange(location: 0, length: abs(stockRowDelta))
    let stockRowChangeIndexSet = NSIndexSet(indexesIn: stockRowChangeRange)
    if stockRowDelta > 0 {
      self.table.insertRows(
        at: stockRowChangeIndexSet,
        withRowType: stockRowControllerIdentifier
      )
    }
    else if stockRowDelta < 0 {
      self.table.removeRows(at: stockRowChangeIndexSet)
    }
}
```

```swift
// Second Approach - Load Stocks Table - insert/remove table rows

func insertRemoveTableRows(stockRowDelta: Int) {
    let stockRowChangeRange = NSRange(location: 0, length: abs(stockRowDelta))
    let stockRowChangeIndexSet = NSIndexSet(indexesIn: stockRowChangeRange)
    if stockRowDelta > 0 {
      self.table.insertRows(
        at: stockRowChangeIndexSet,
        withRowType: stockRowControllerIdentifier
      )
    }
    else if stockRowDelta < 0 {
      self.table.removeRows(at: stockRowChangeIndexSet)
    }
}
```

```
// Second Approach - Load Stocks Table - insert/remove table rows

func insertRemoveTableRows(stockRowDelta: Int) {
    let stockRowChangeRange = NSRange(location: 0, length: abs(stockRowDelta))
    let stockRowChangeIndexSet = NSIndexSet(indexesIn: stockRowChangeRange)
    if stockRowDelta > 0 {
      self.table.insertRows(
        at: stockRowChangeIndexSet,
        withRowType: stockRowControllerIdentifier
      )
    }
    else if stockRowDelta < 0 {
      self.table.removeRows(at: stockRowChangeIndexSet)
    }
}
```

```
// Second Approach - Load Stocks Table - insert/remove table rows

func insertRemoveTableRows(stockRowDelta: Int) {
    let stockRowChangeRange = NSRange(location: 0, length: abs(stockRowDelta))
    let stockRowChangeIndexSet = NSIndexSet(indexesIn: stockRowChangeRange)
    if stockRowDelta > 0 {
      self.table.insertRows(
        at: stockRowChangeIndexSet,
        withRowType: stockRowControllerIdentifier
      )
    }
    else if stockRowDelta < 0 {
      self.table.removeRows(at: stockRowChangeIndexSet)
    }
}
```

```swift
// Second Approach - Load Stocks Table - insert/remove table rows

func insertRemoveTableRows(stockRowDelta: Int) {
    let stockRowChangeRange = NSRange(location: 0, length: abs(stockRowDelta))
    let stockRowChangeIndexSet = NSIndexSet(indexesIn: stockRowChangeRange)
    if stockRowDelta > 0 {
      self.table.insertRows(
        at: stockRowChangeIndexSet,
        withRowType: stockRowControllerIdentifier
      )
    }
    else if stockRowDelta < 0 {
      self.table.removeRows(at: stockRowChangeIndexSet)
    }
}
```

```
// Second Approach - Load Stocks Table - insert/remove table rows

func insertRemoveTableRows(stockRowDelta: Int) {
    let stockRowChangeRange = NSRange(location: 0, length: abs(stockRowDelta))
    let stockRowChangeIndexSet = NSIndexSet(indexesIn: stockRowChangeRange)
    if stockRowDelta > 0 {
      self.table.insertRows(
        at: stockRowChangeIndexSet,
        withRowType: stockRowControllerIdentifier
      )
    }
    else if stockRowDelta < 0 {
      self.table.removeRows(at: stockRowChangeIndexSet)
    }
}
```

Resume Time

Improve WKInterfaceTable loading performance

Resume Time

Improve WKInterfaceTable loading performance

Number of stocks in list is capped

Resume Time

Improve WKInterfaceTable loading performance

Number of stocks in list is capped

Inserting/removing rows is much more efficient than reloading the entire table

Resume Time

Improve WKInterfaceTable loading performance

Resume Time

Improve WKInterfaceTable loading performance

Instead of iterating over entire table when single rows are updated, consider:

Resume Time

Improve WKInterfaceTable loading performance

Instead of iterating over entire table when single rows are updated, consider:

- Use `rowController(at index: Int) -> AnyObject?` to get RowController to be updated

Resume Time

Improve WKInterfaceTable loading performance

Instead of iterating over entire table when single rows are updated, consider:

- Use `rowController(at index: Int) -> AnyObject?` to get RowController to be updated
- Store a reference to the RowController to update later

Resume Time

Updating your UI elements

WKInterfaceObjects are modified in the extension process

Updates to these properties are sent from extension process to app process

App process handles layout of interface

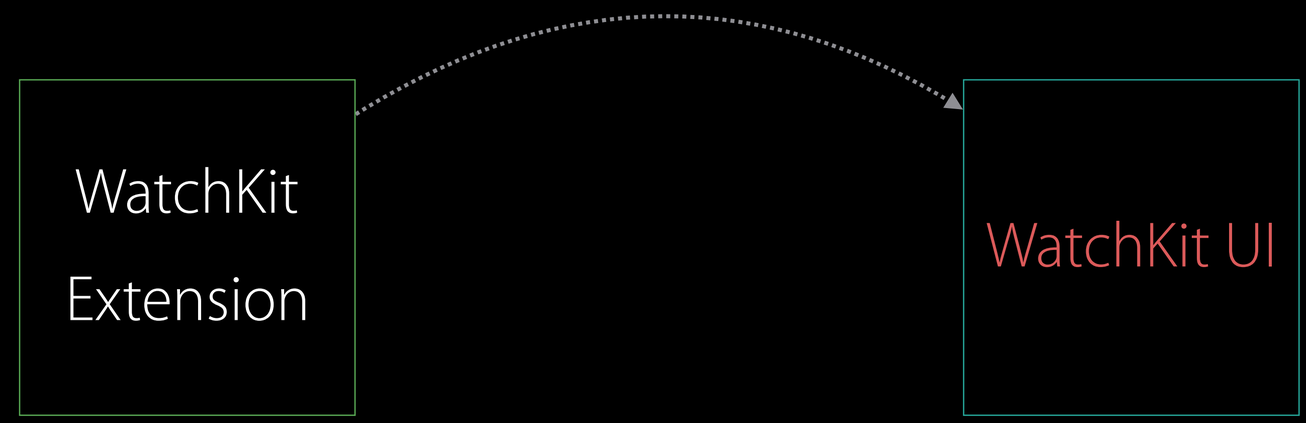
Resume Time

Updating your UI elements

WKInterfaceObjects are modified in the extension process

Updates to these properties are sent from extension process to app process

App process handles layout of interface



Resume Time

StocksInterfaceController layout

```
@IBOutlet weak var platter: WKInterfaceGroup!
@IBOutlet weak var listNameLabel: WKInterfaceLabel!
@IBOutlet weak var changeInPointsLabel: WKInterfaceLabel!
@IBOutlet weak var priceLabel: WKInterfaceLabel!
```

Resume Time

StocksInterfaceController layout

```
@IBOutlet weak var platter: WKInterfaceGroup!
@IBOutlet weak var listNameLabel: WKInterfaceLabel!
@IBOutlet weak var changeInPointsLabel: WKInterfaceLabel!
@IBOutlet weak var priceLabel: WKInterfaceLabel!
```

Resume Time

StocksInterfaceController layout

```
@IBOutlet weak var platter: WKInterfaceGroup!
@IBOutlet weak var listNameLabel: WKInterfaceLabel!
@IBOutlet weak var changeInPointsLabel: WKInterfaceLabel!
@IBOutlet weak var priceLabel: WKInterfaceLabel!
```

Resume Time

StocksInterfaceController layout

```
@IBOutlet weak var platter: WKInterfaceGroup!
@IBOutlet weak var listNameLabel: WKInterfaceLabel!
@IBOutlet weak var changeInPointsLabel: WKInterfaceLabel!
@IBOutlet weak var priceLabel: WKInterfaceLabel!
```

Resume Time

StocksInterfaceController layout

```
@IBOutlet weak var platter: WKInterfaceGroup!
@IBOutlet weak var listNameLabel: WKInterfaceLabel!
@IBOutlet weak var changeInPointsLabel: WKInterfaceLabel!
@IBOutlet weak var priceLabel: WKInterfaceLabel!
```

Resume Time

StocksInterfaceController layout

```
@IBOutlet weak var platter: WKInterfaceGroup!
@IBOutlet weak var listNameLabel: WKInterfaceLabel!
@IBOutlet weak var changeInPointsLabel: WKInterfaceLabel!
@IBOutlet weak var priceLabel: WKInterfaceLabel!
```

Resume Time

StocksInterfaceController layout

```
@IBOutlet weak var platter: WKInterfaceGroup!
@IBOutlet weak var listNameLabel: WKInterfaceLabel!
@IBOutlet weak var changeInPointsLabel: WKInterfaceLabel!
@IBOutlet weak var priceLabel: WKInterfaceLabel!
```

Resume Time

StocksInterfaceController layout

```
@IBOutlet weak var platter: WKInterfaceGroup!
@IBOutlet weak var listNameLabel: WKInterfaceLabel!
@IBOutlet weak var changeInPointsLabel: WKInterfaceLabel!
@IBOutlet weak var priceLabel: WKInterfaceLabel!
```

```
// Initial Approach - Update StockRowController

func update(listName: String, price: String, changeInPoints: String,
            changeLabelColor: UIColor, platterColor: UIColor) {
    self.platter.setBackgroundColor(platterColor)
    self.listNameLabel.setText(listName)
    self.changeInPointsLabel.setText(changeInPoints)
    self.changeInPointsLabel.setTextColor(changeLabelColor)
    self.priceLabel.setText(price)
}
```

Resume Time

Improve layout update performance

Resume Time

Improve layout update performance

Properties on WKInterfaceObject are not cached

Resume Time

Improve layout update performance

Properties on WKInterfaceObject are not cached

Setting a property on WKInterfaceObject sends that value to the app process every time

Resume Time

Improve layout update performance

Properties on WKInterfaceObject are not cached

Setting a property on WKInterfaceObject sends that value to the app process every time

On average, 200ms for value to move from extension to app process

Resume Time

Improve layout update performance

Properties on WKInterfaceObject are not cached

Setting a property on WKInterfaceObject sends that value to the app process every time

On average, 200ms for value to move from extension to app process

1.4s worst case scenario when initially loading the stocks list

```
// Better Approach - Update StockRowController Part 1

func update(listName: String, price: String, changeInPoints: String, changeLabelColor:
UIColor, platterColor: UIColor) {
    if self.platterColor?.hash != platterColor.hash {
      self.platterColor = platterColor
      self.platter.setBackgroundColor(platterColor)
    }
    if self.listName?.hash != listName.hash {
      self.listName = listName
      self.listNameLabel.setText(listName)
    }
    // ...
}
```

```
// Better Approach - Update StockRowController Part 1

func update(listName: String, price: String, changeInPoints: String, changeLabelColor:
UIColor, platterColor: UIColor) {
    if self.platterColor?.hash != platterColor.hash {
      self.platterColor = platterColor
      self.platter.setBackgroundColor(platterColor)
    }
    if self.listName?.hash != listName.hash {
      self.listName = listName
      self.listNameLabel.setText(listName)
    }
    // ...
}
```

```
// Better Approach - Update StockRowController Part 1

func update(listName: String, price: String, changeInPoints: String, changeLabelColor:
UIColor, platterColor: UIColor) {
    if self.platterColor?.hash != platterColor.hash {
      self.platterColor = platterColor
      self.platter.setBackgroundColor(platterColor)
    }
    if self.listName?.hash != listName.hash {
      self.listName = listName
      self.listNameLabel.setText(listName)
    }
    // ...
}
```

```
// Better Approach - Update StockRowController Part 1

func update(listName: String, price: String, changeInPoints: String, changeLabelColor:
UIColor, platterColor: UIColor) {
    if self.platterColor?.hash != platterColor.hash {
      self.platterColor = platterColor
      self.platter.setBackgroundColor(platterColor)
    }
    if self.listName?.hash != listName.hash {
      self.listName = listName
      self.listNameLabel.setText(listName)
    }
    // ...
}
```

Stocks

Resume time recap

Stocks

Resume time recap

Minimize work performed `willActivate` and `didAppear`

Stocks

Resume time recap

Minimize work performed `willActivate` and `didAppear`

Make use of cancelable operations

Stocks

Resume time recap

Minimize work performed `willActivate` and `didAppear`

Make use of cancelable operations

Overly complicated user interfaces will lead to slower resume times

Stocks

Resume time recap

Minimize work performed `willActivate` and `didAppear`

Make use of cancelable operations

Overly complicated user interfaces will lead to slower resume times

Only update your user interface when necessary

Summary

Summary

Think small

- Keep tasks small and easy to perform
- Simplify your user interface
- Make use of new Background Refresh APIs

Summary

Think small

- Keep tasks small and easy to perform
- Simplify your user interface
- Make use of new Background Refresh APIs

Focus on resume time

- Pay attention to `WKInterfaceController` lifecycle methods (especially `willActivate` and `didAppear`)
- Make use of cancelable operations
- Optimize when updating your user interface

More Information

https://developer.apple.com/wwdc16/227

Related Sessions

What's New in watchOS 3	Presidio	Tuesday 5:00PM
Quick Interaction Techniques for watchOS	Presidio	Wednesday 11:00AM
Designing Great Apple Watch Experiences	Presidio	Wednesday 1:40PM
Keeping Your Watch App Up to Date	Mission	Thursday 9:00AM
Concurrent Programming With GCD in Swift 3	Pacific Heights	Friday 4:00PM
Advanced NSOperations		WWDC 2015

Labs

WatchKit and WatchConnectivity Lab	Frameworks Lab B Friday 2:00PM

WWDC16