System Frameworks

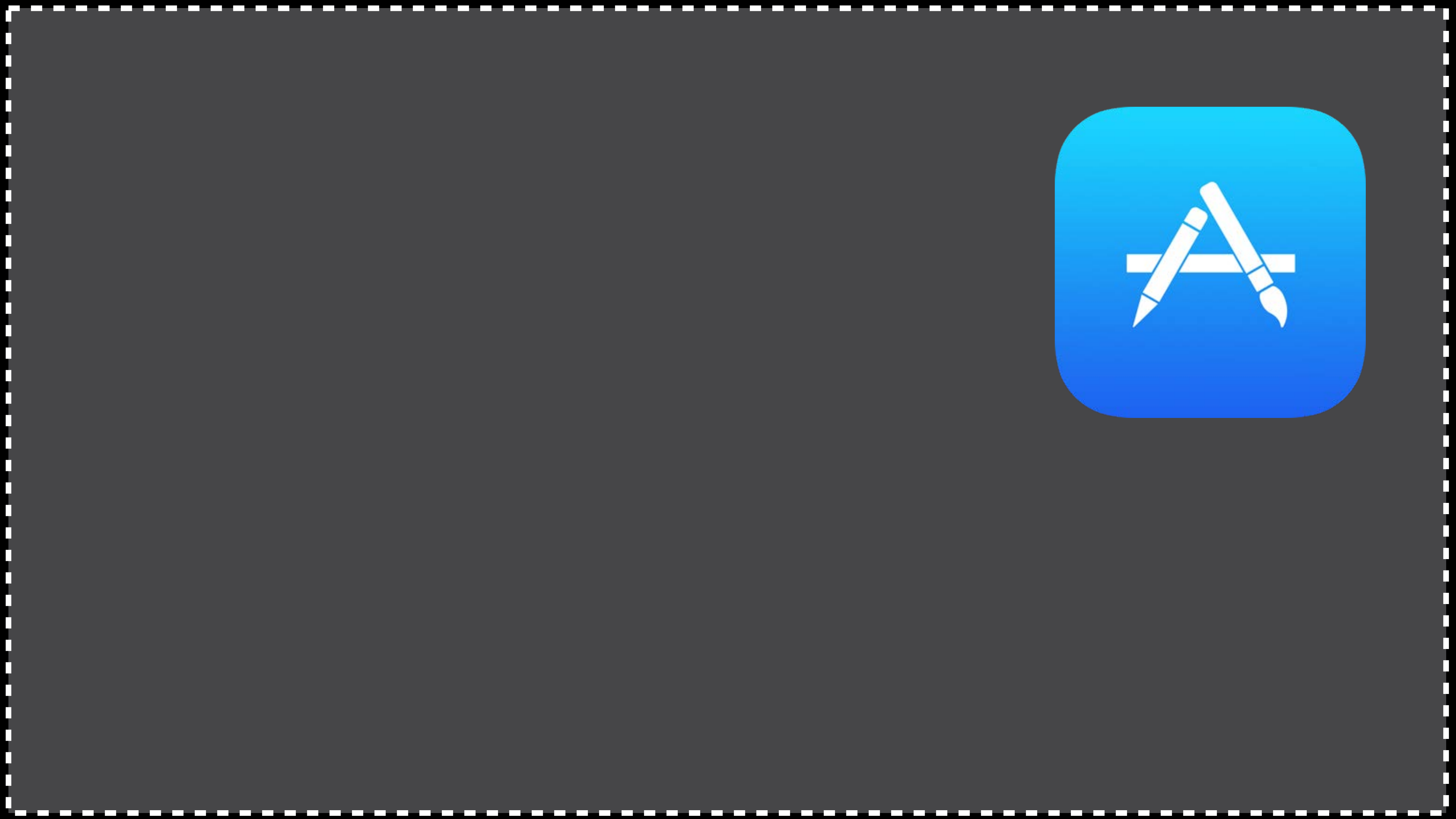
#WWDC16

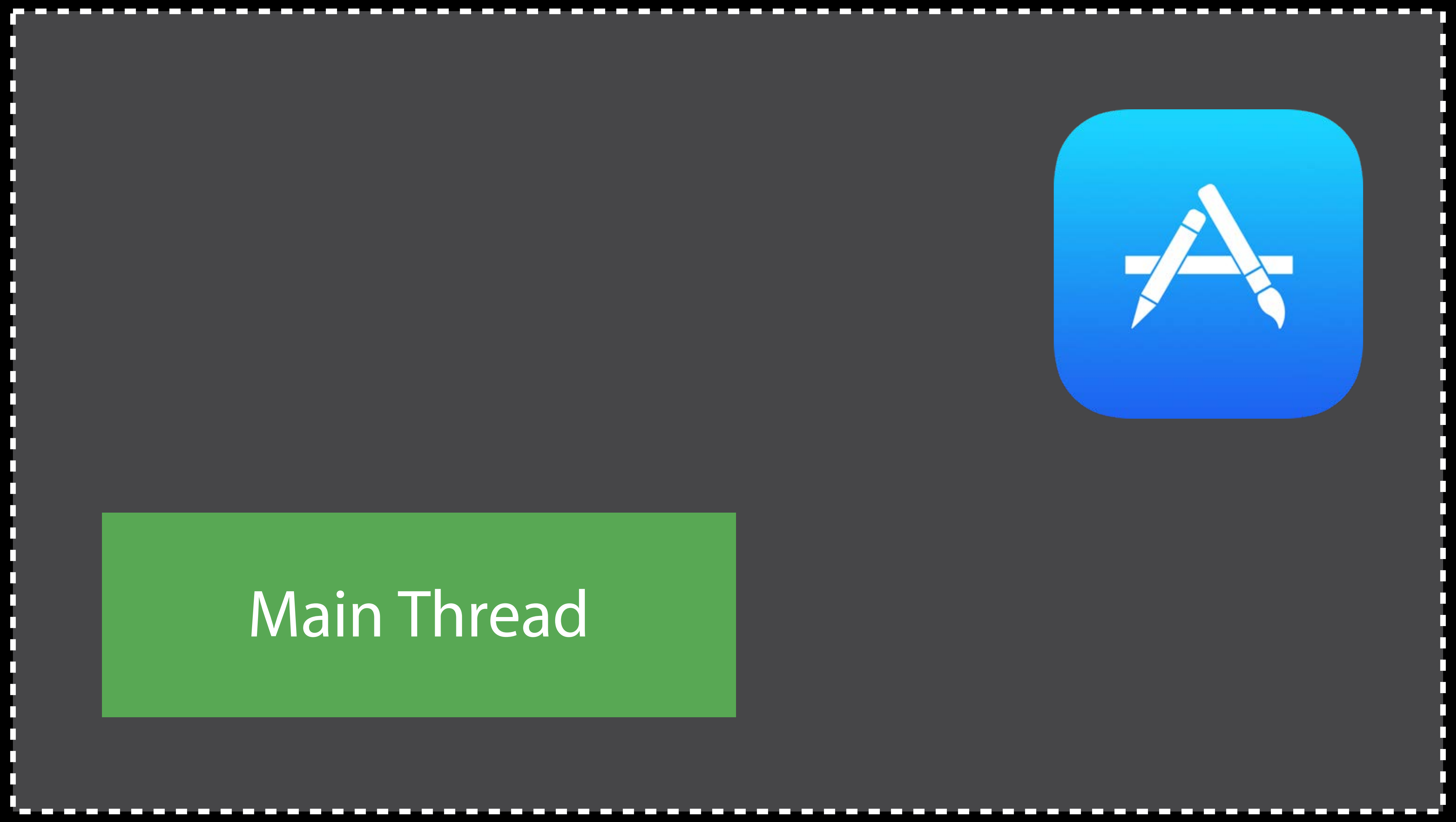
Concurrent Programming with GCD in Swift 3

Session 720

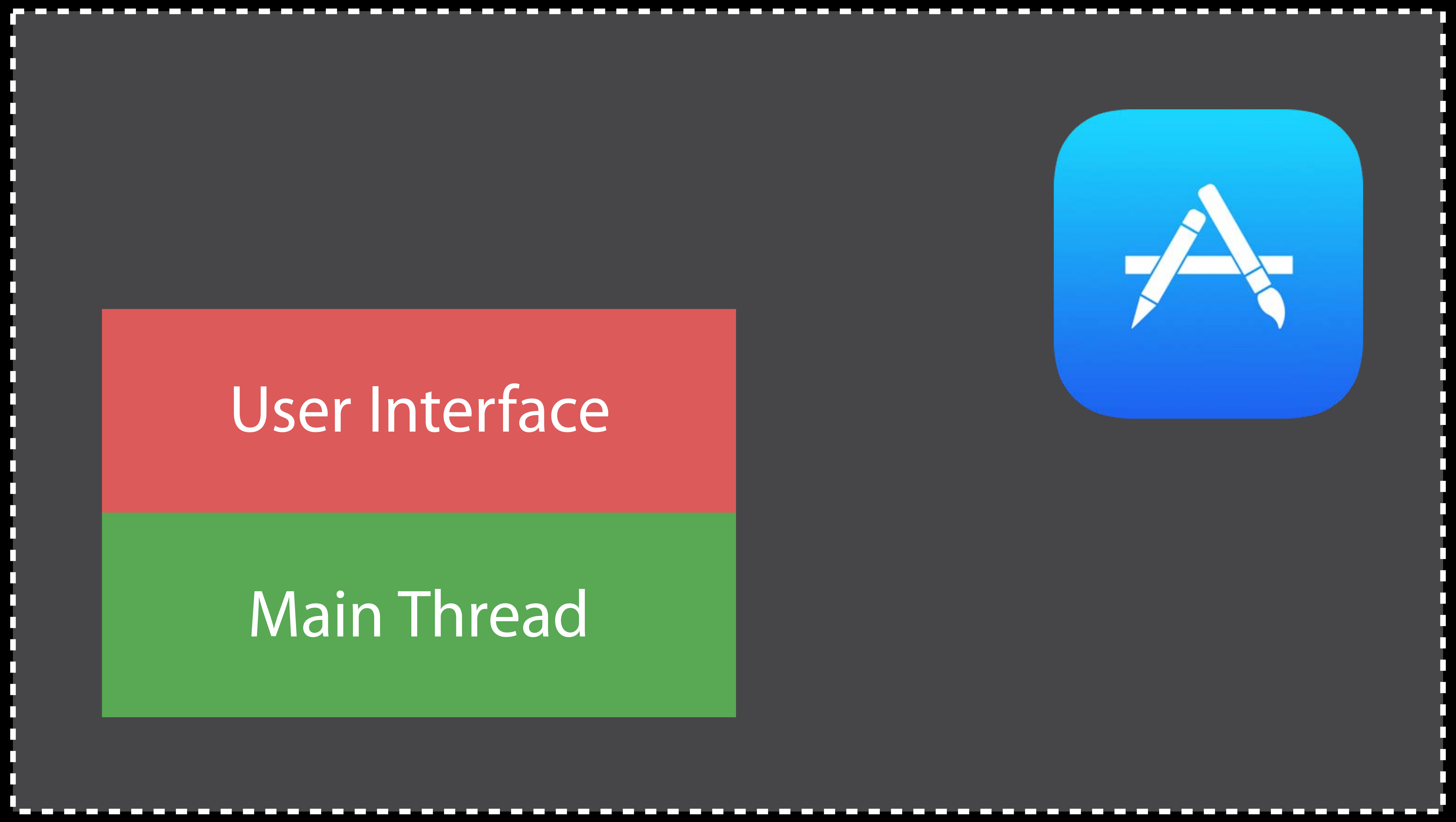
Matt Wright Darwin Runtime Engineer
Pierre Habouzit Darwin Runtime Engineer

© 2016 Apple Inc. All rights reserved. Redistribution or public display not permitted without written permission from Apple.



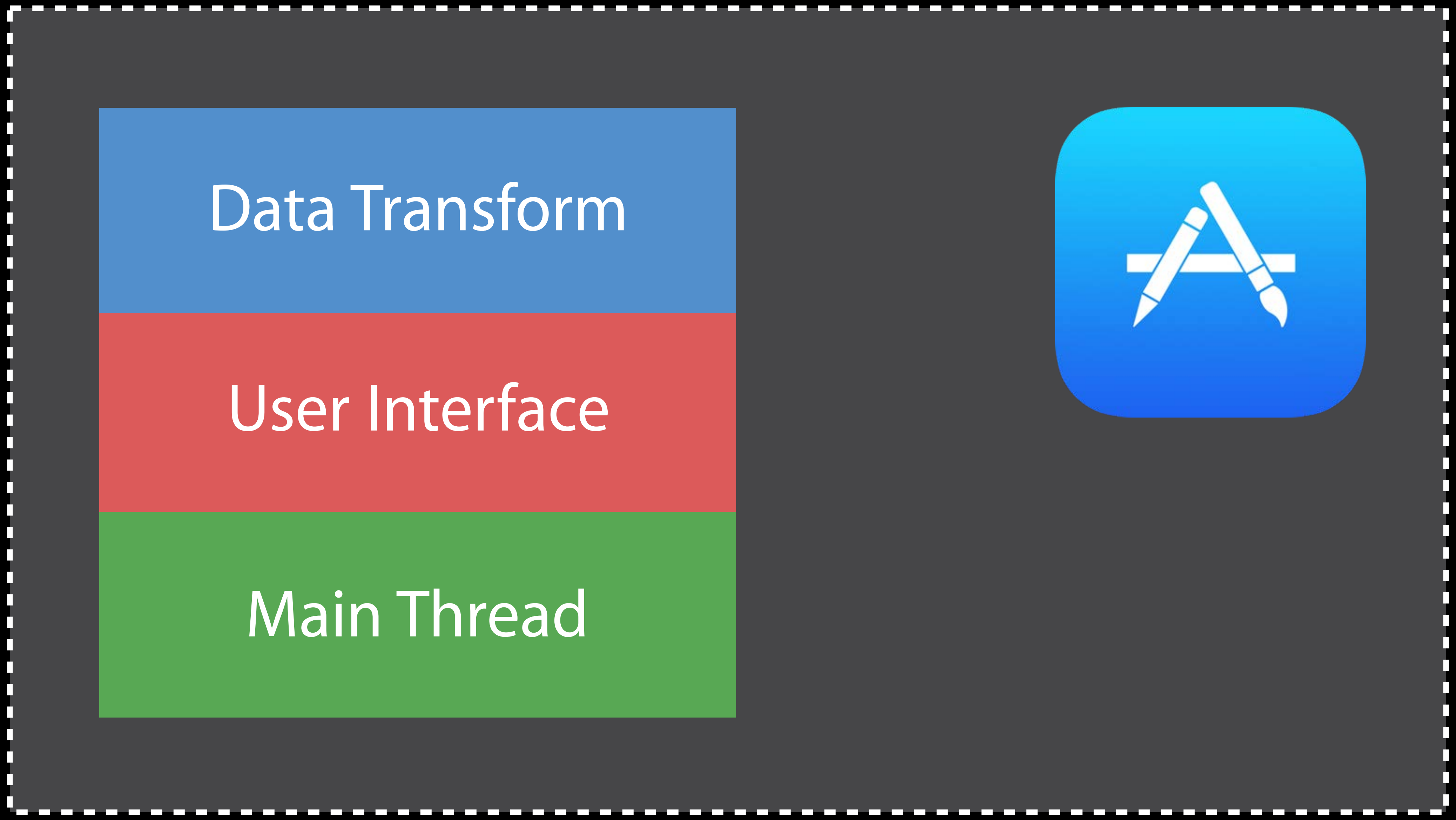


Main Thread



User Interface

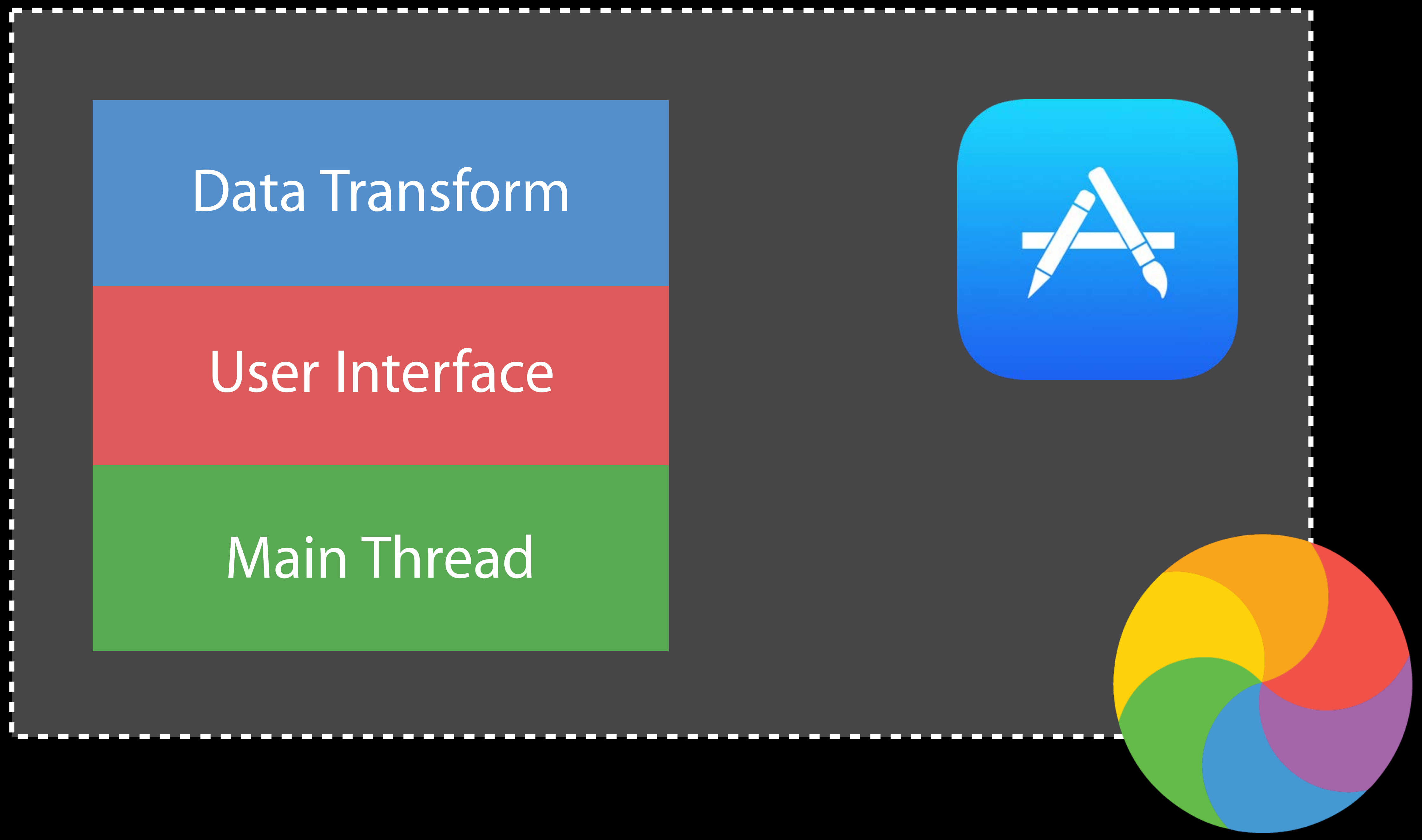
Main Thread



Data Transform

User Interface

Main Thread



Data Transform

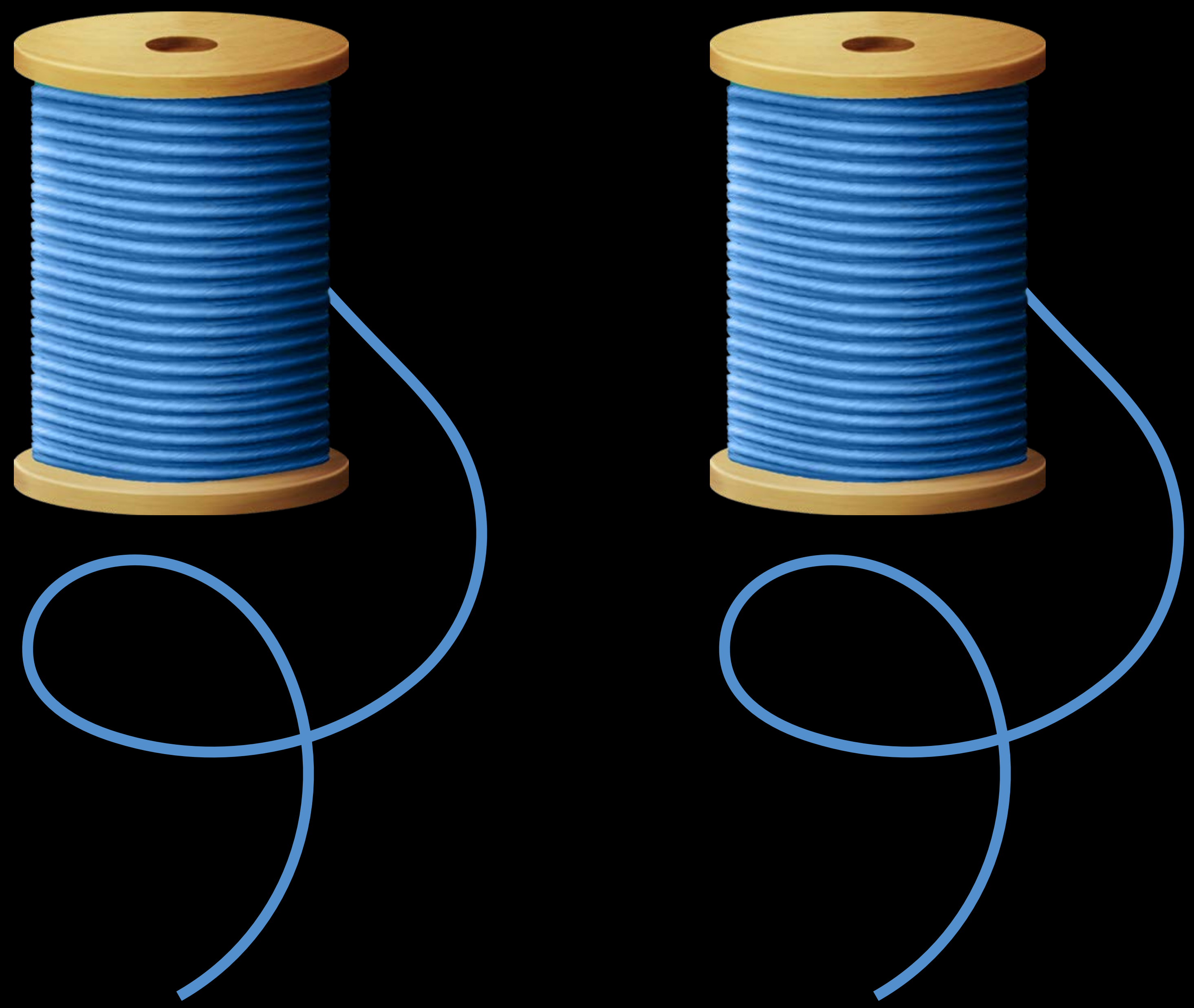
User Interface

Main Thread

Concurrency

Concurrency

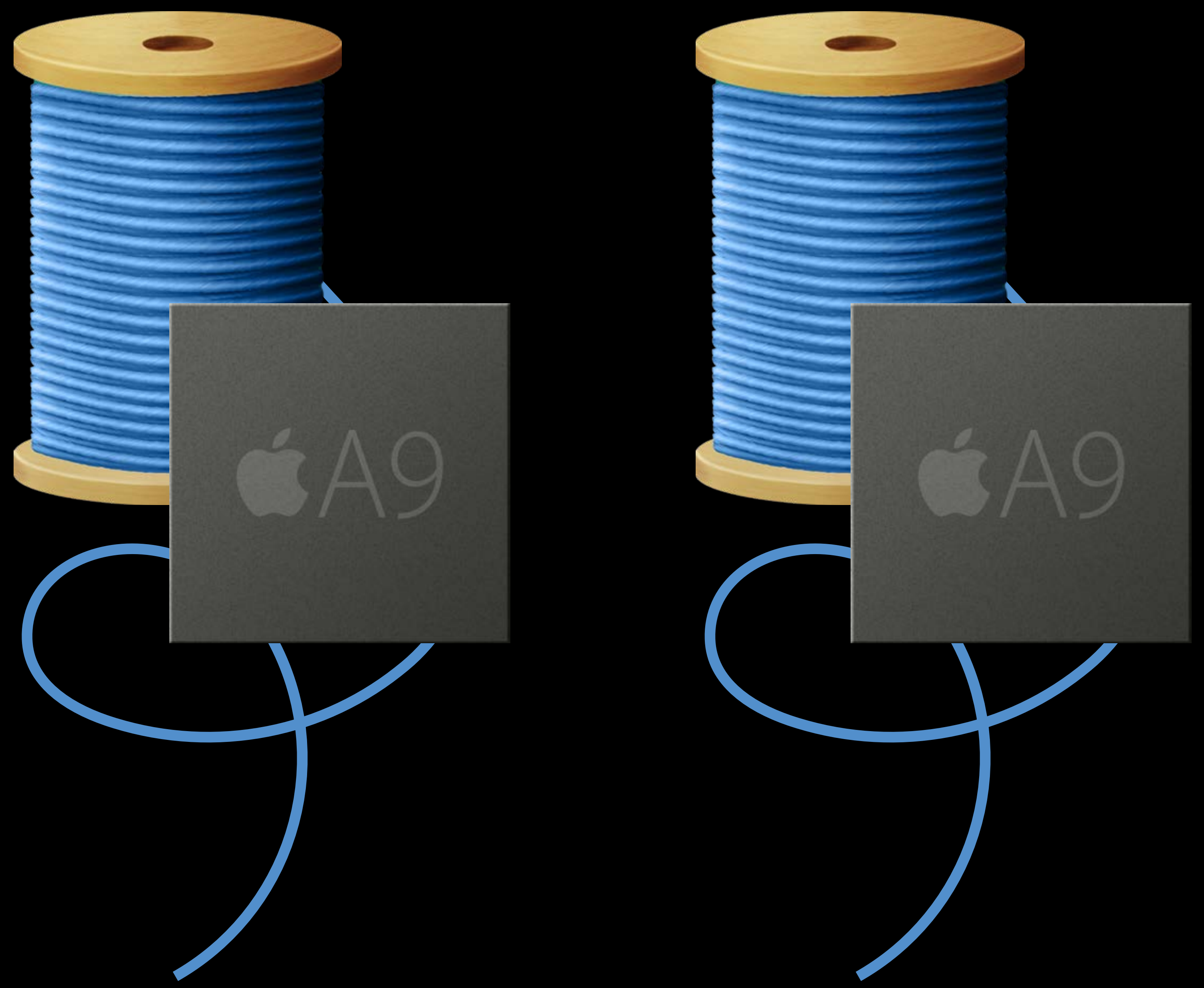
Threads allow execution of code at the same time



Concurrency

Threads allow execution of code at the same time

CPU cores can each execute a single thread at any given time

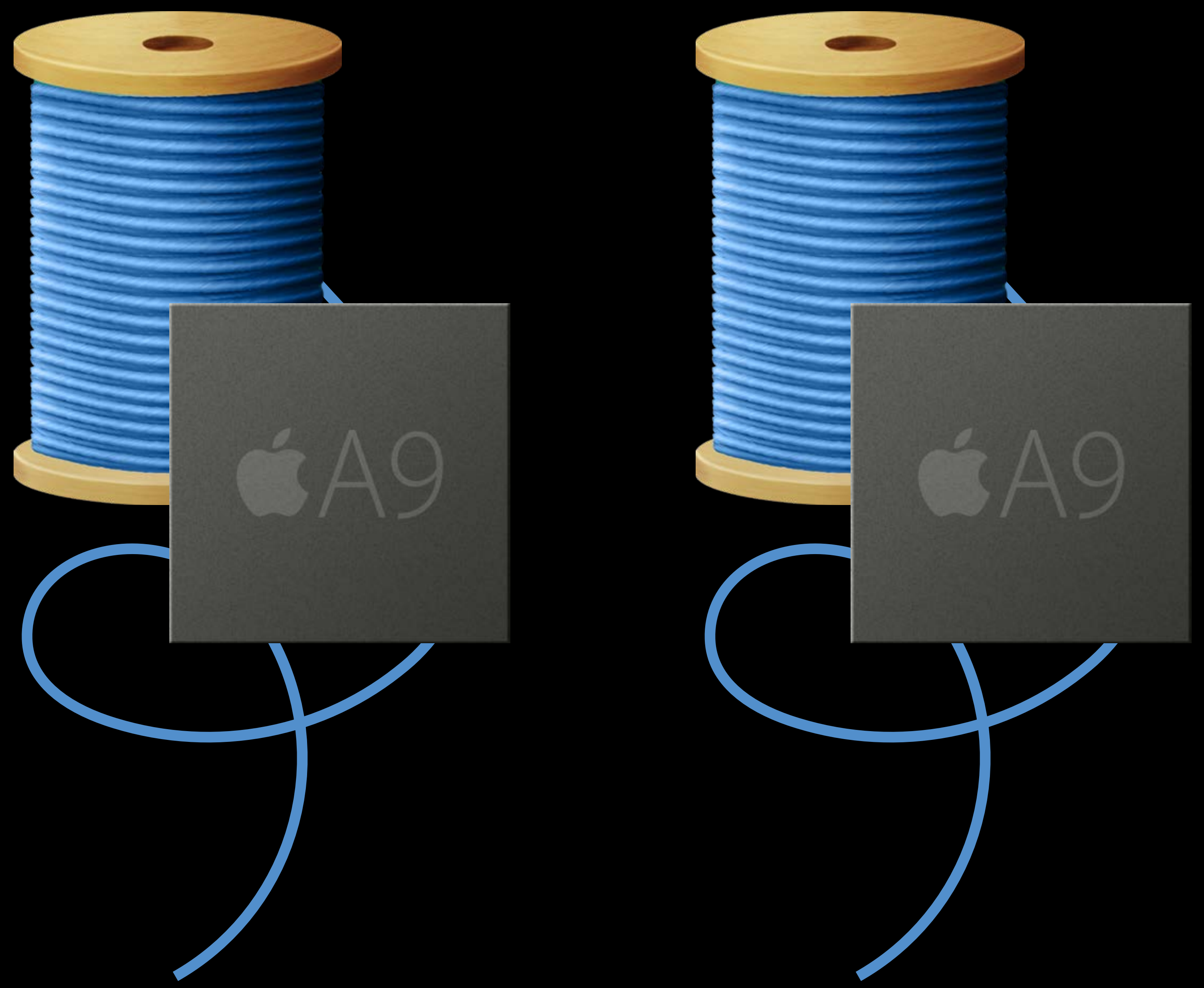


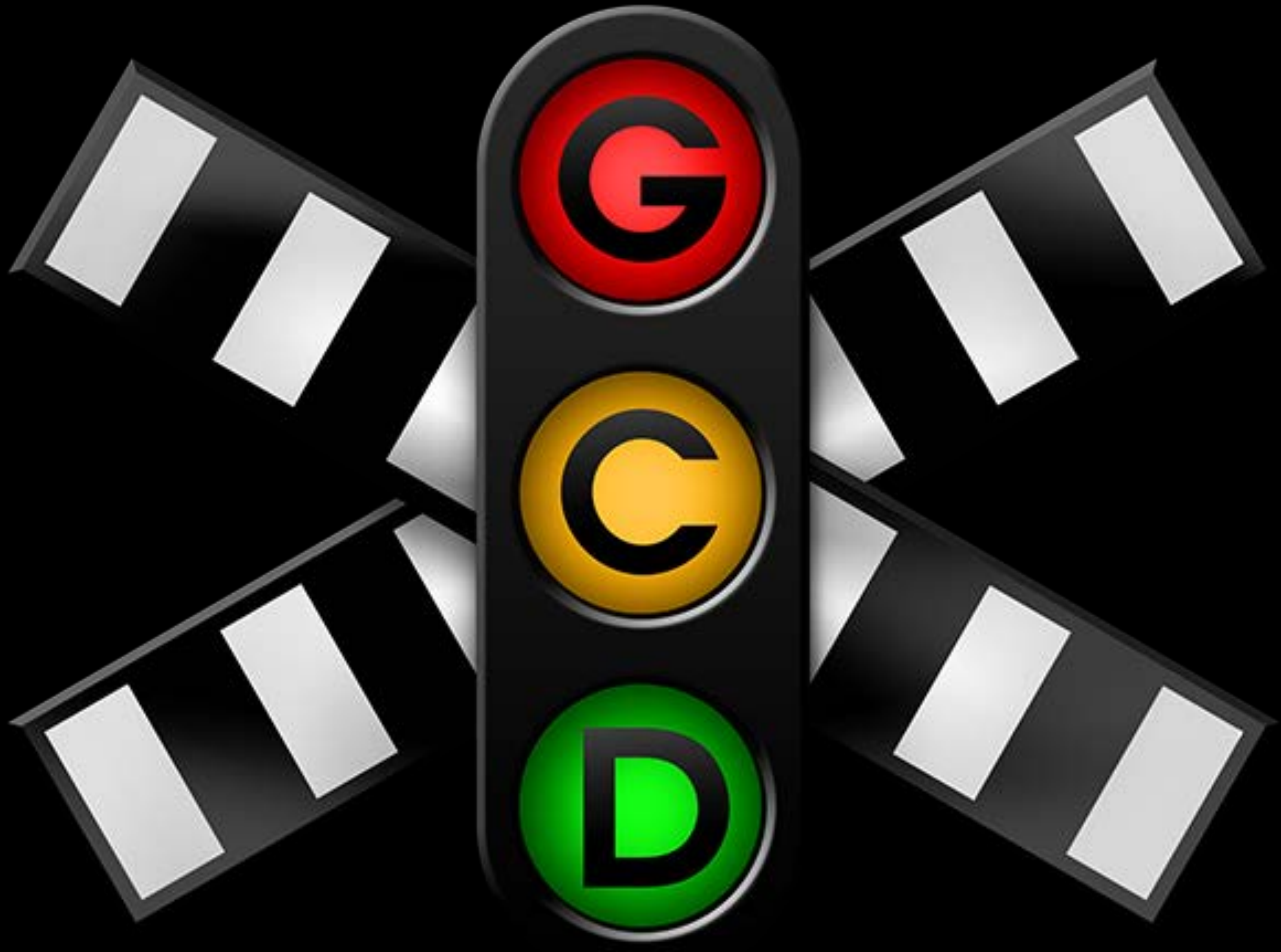
Concurrency

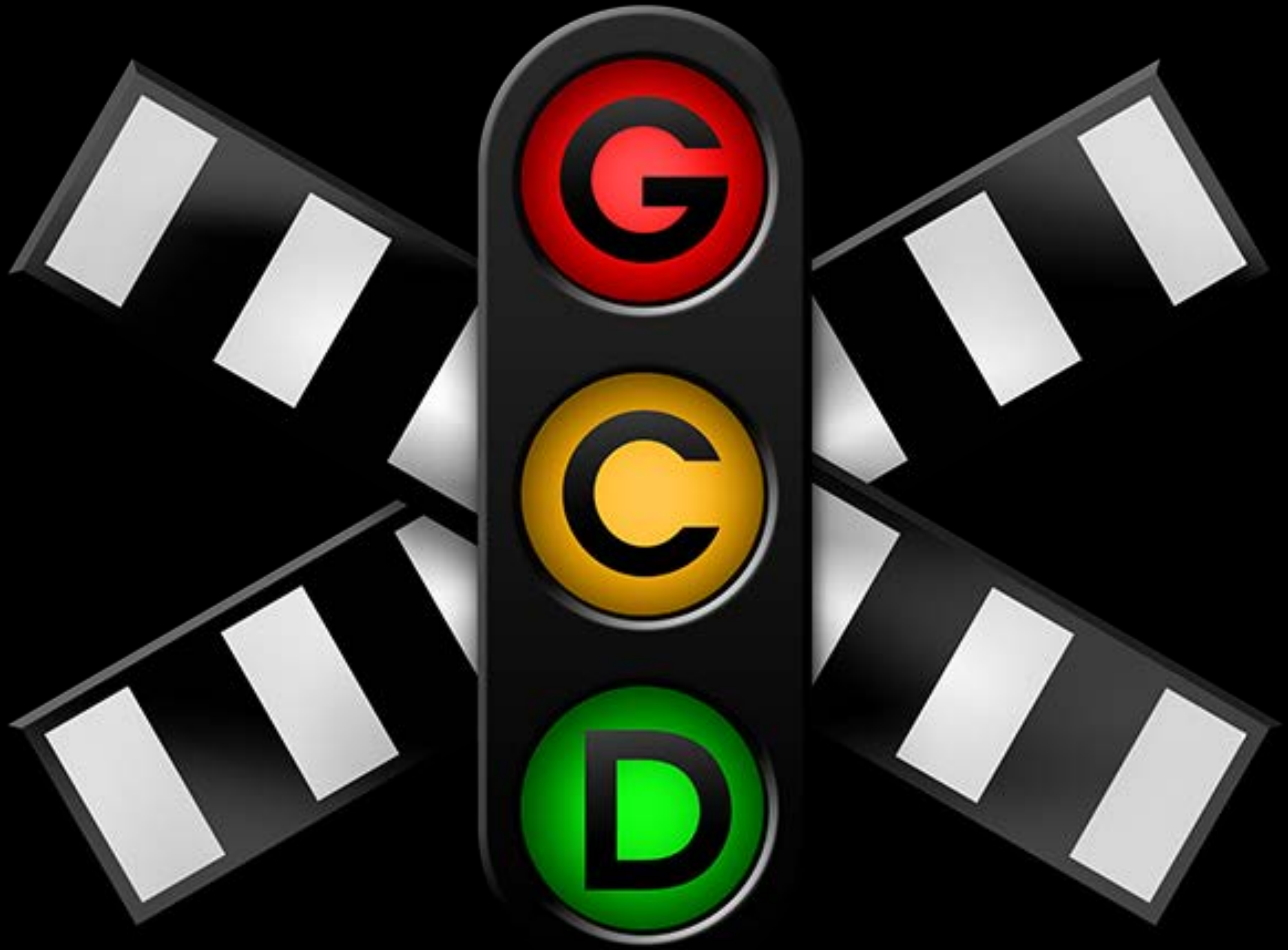
Threads allow execution of code at the same time

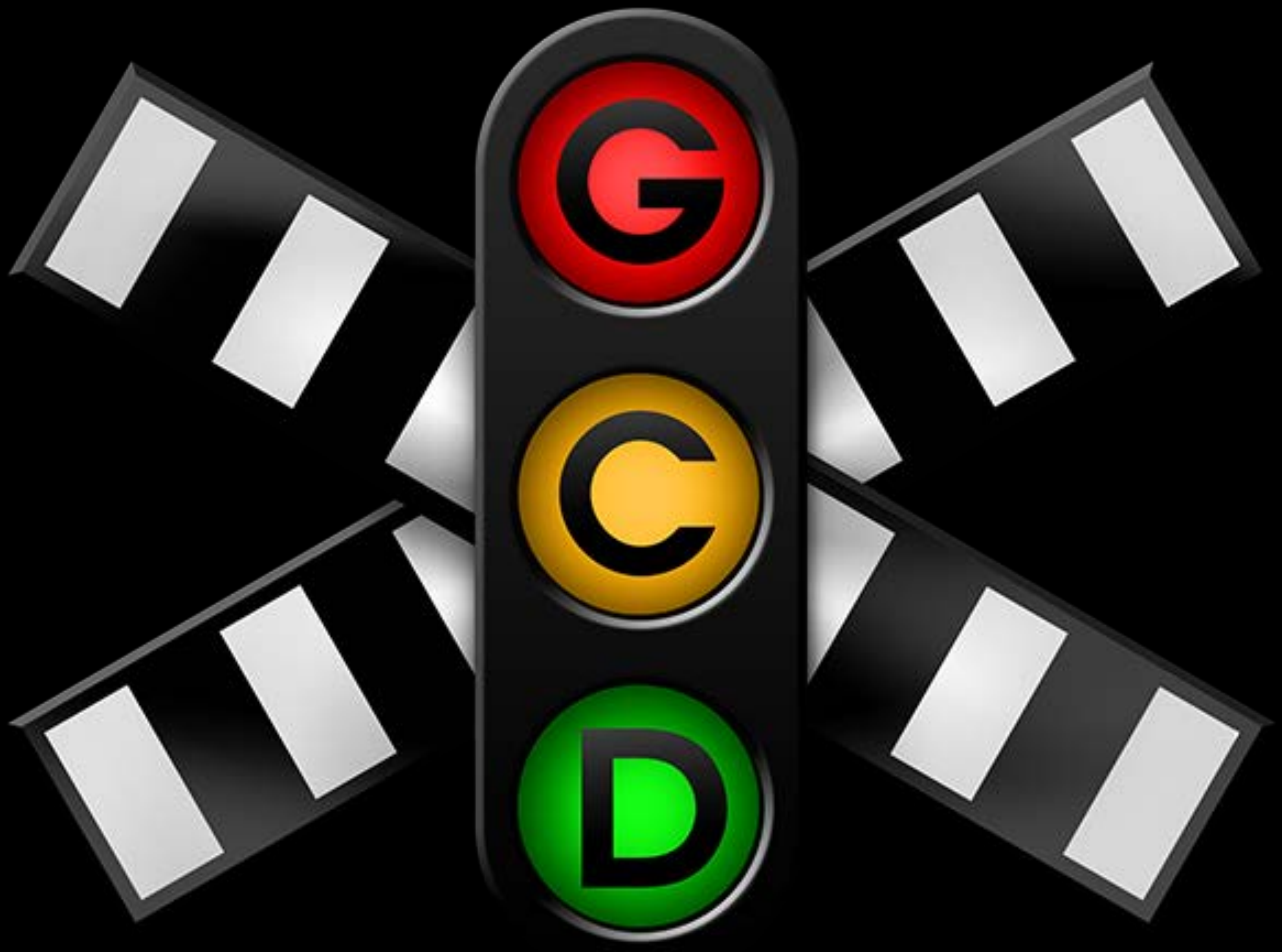
CPU cores can each execute a single thread at any given time

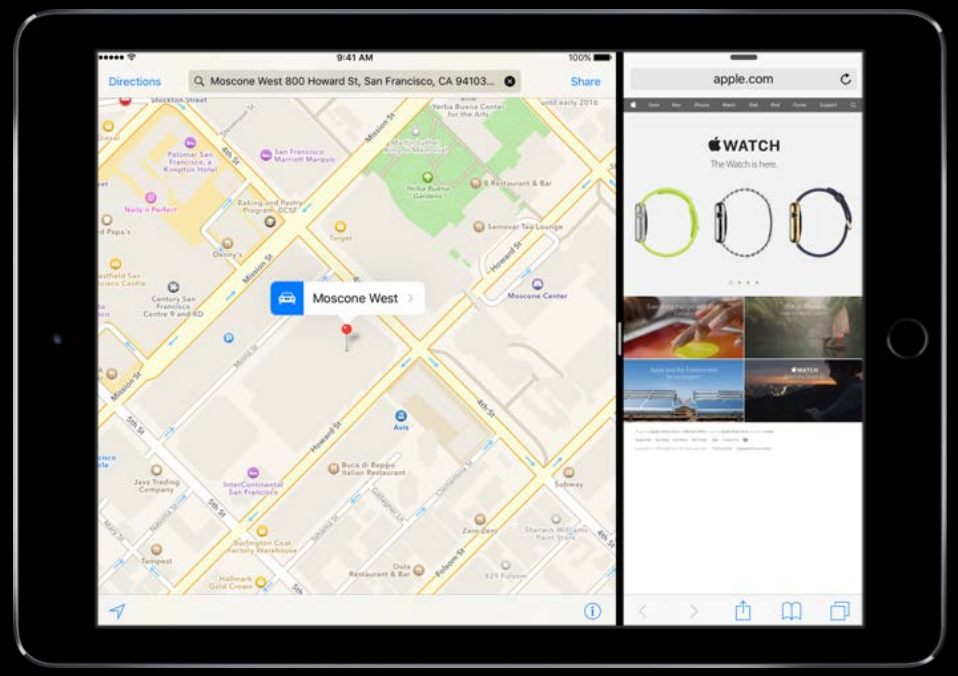
Maintaining code invariants is more difficult with concurrency

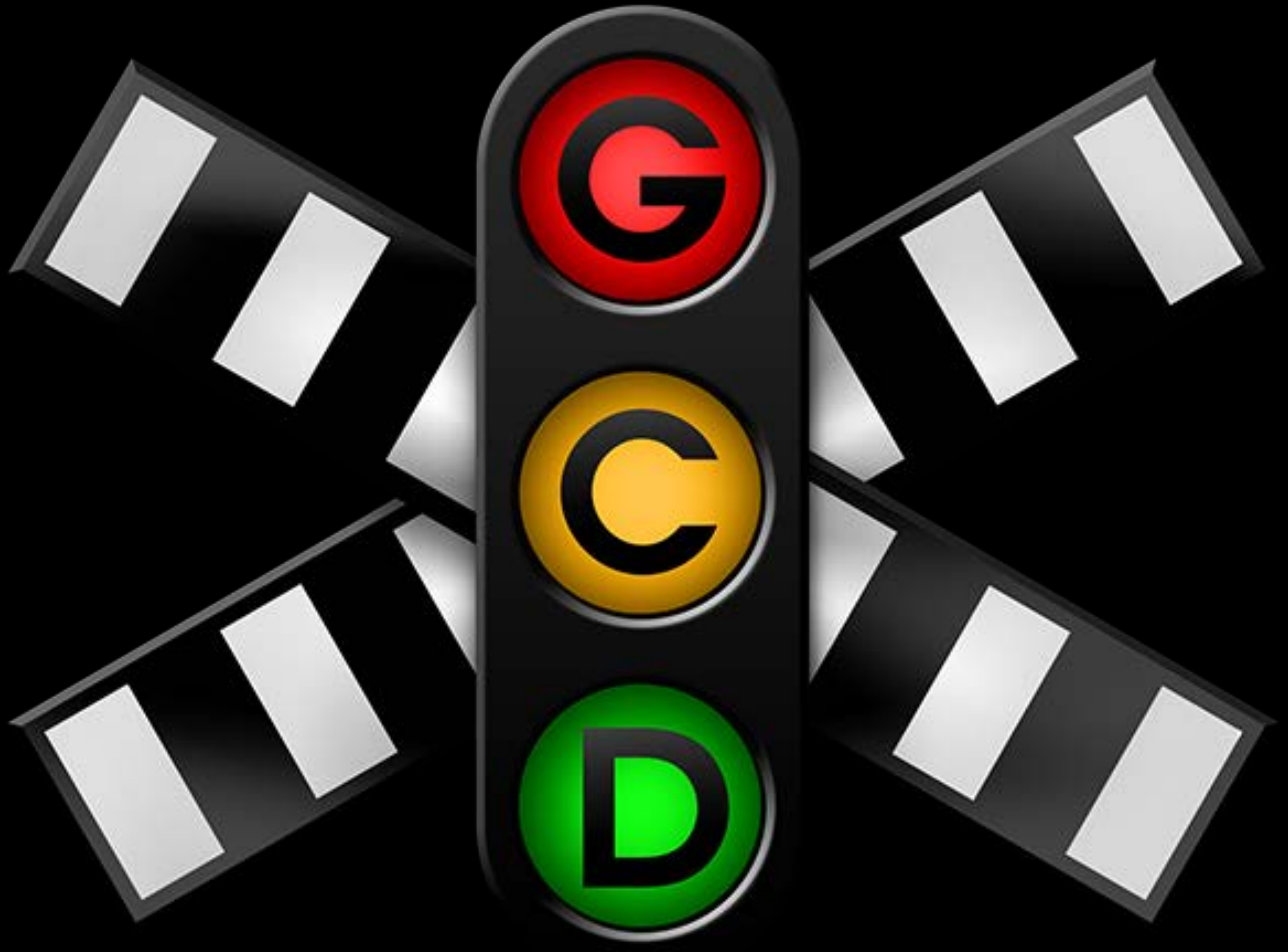


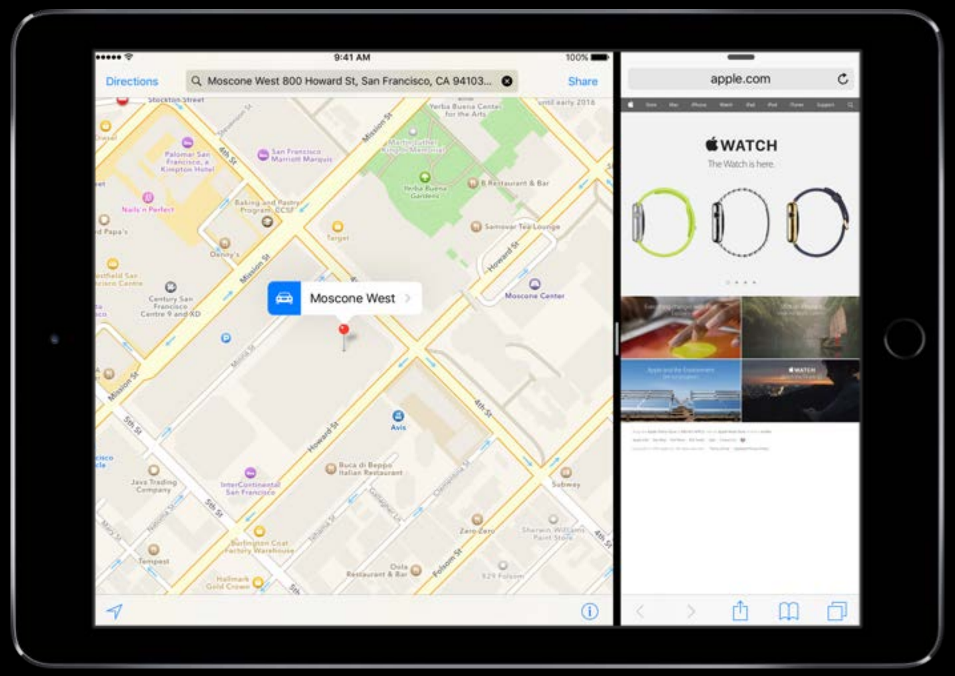

G
C
D

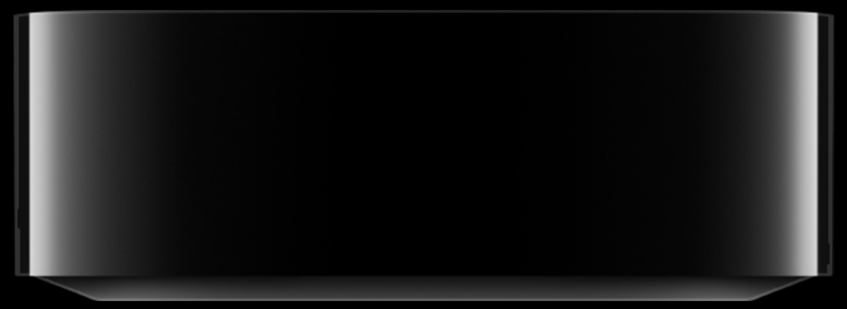

G
C
D

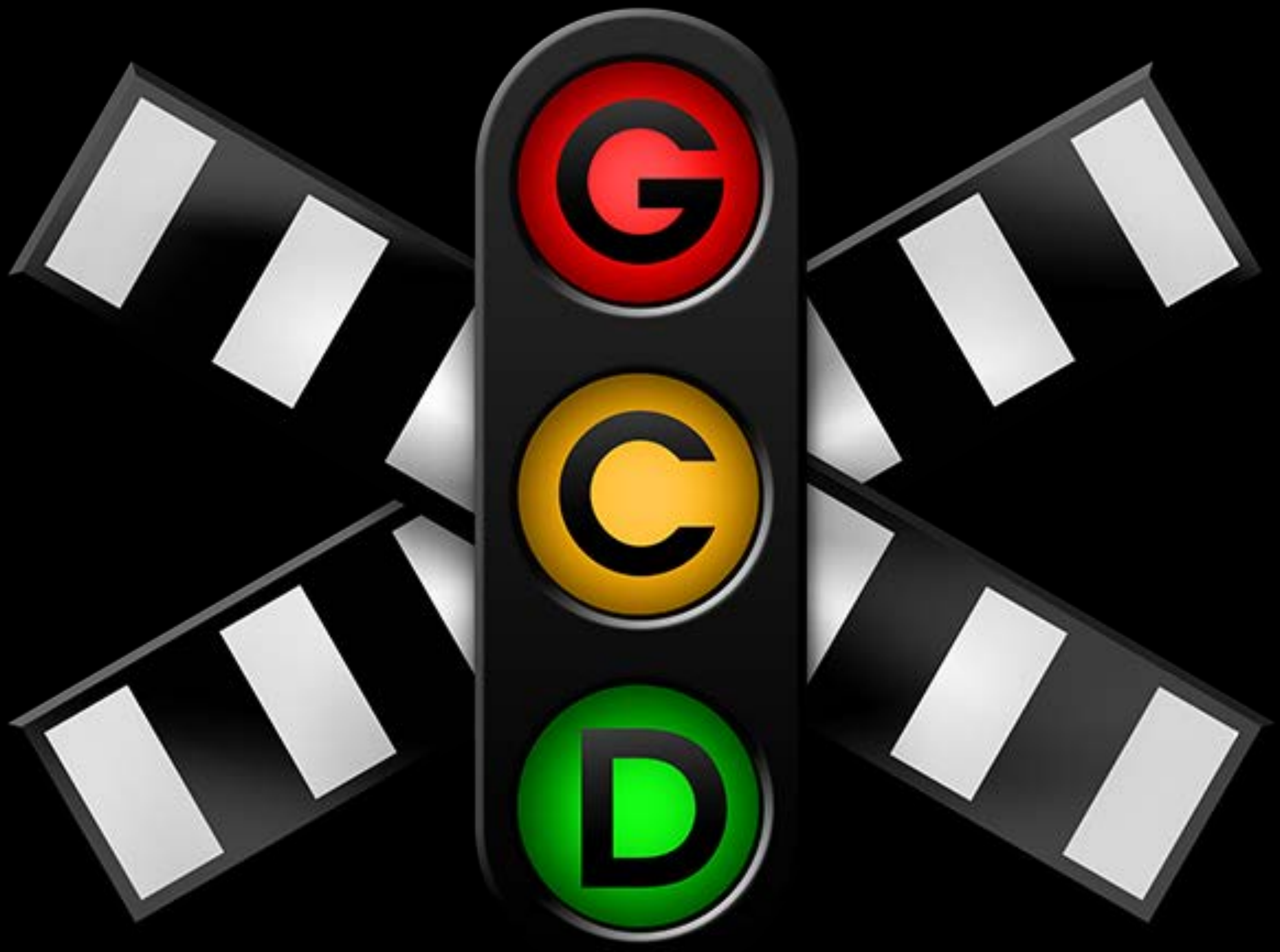

G
C
D

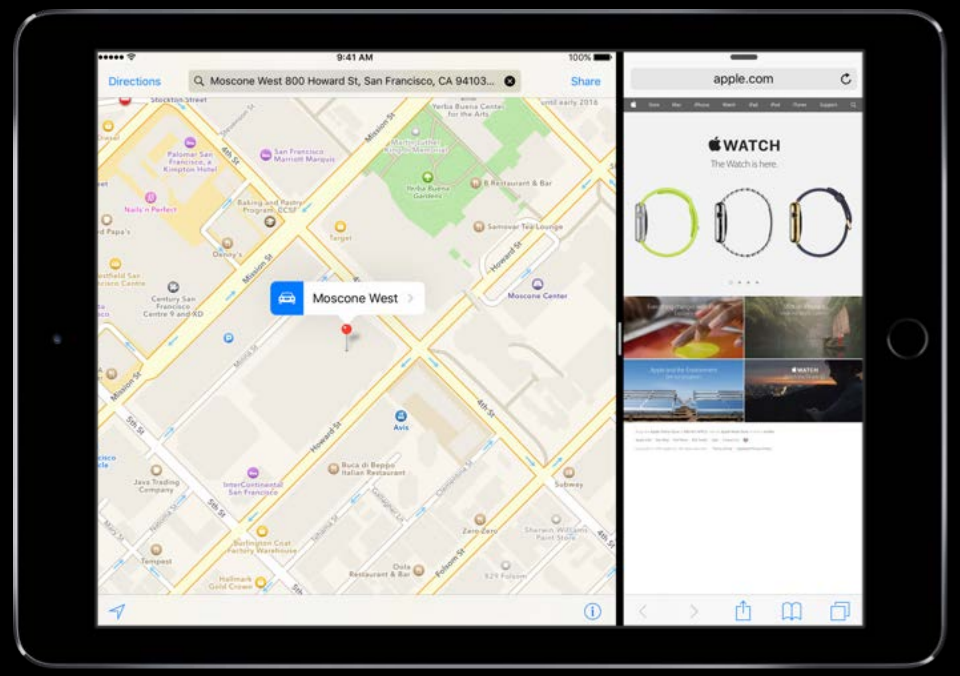

Directions
Share
Moscone West
apple.com
WATCH

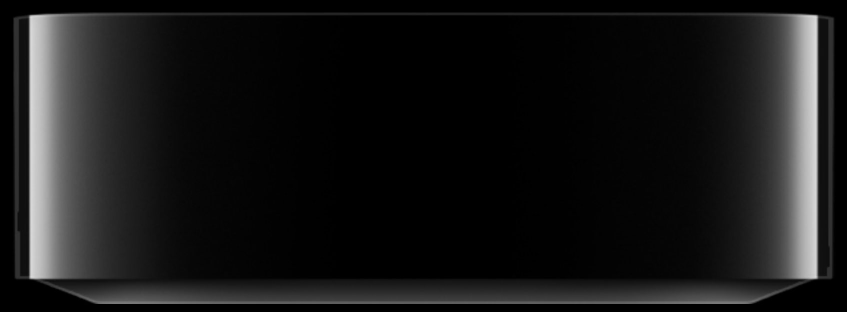

G
C
D


Moscone West
apple.com
WATCH




G
C
D


Moscone West
apple.com
WATCH

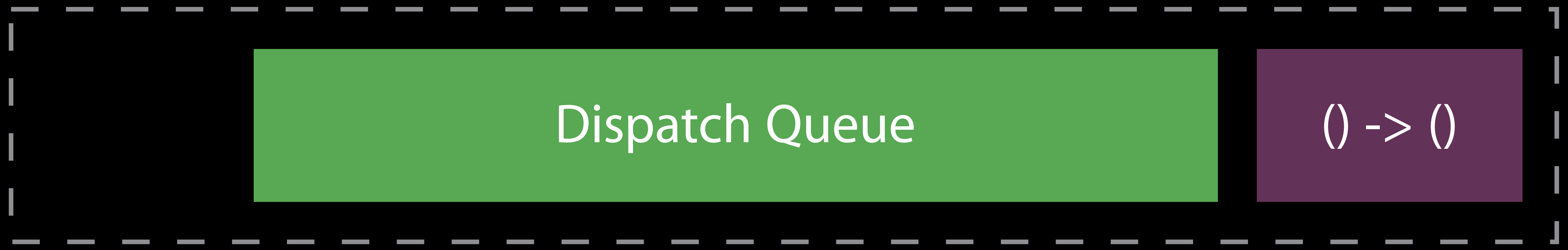


Dispatch Queues and Run Loops

Dispatch Queues and Run Loops

Dispatch Queue

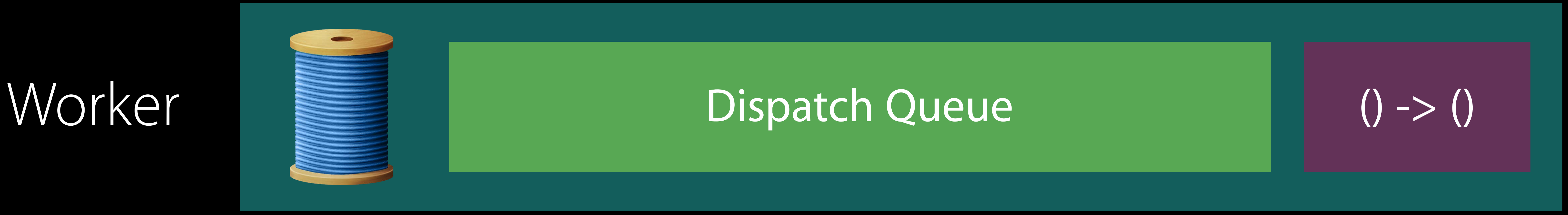
Dispatch Queues and Run Loops



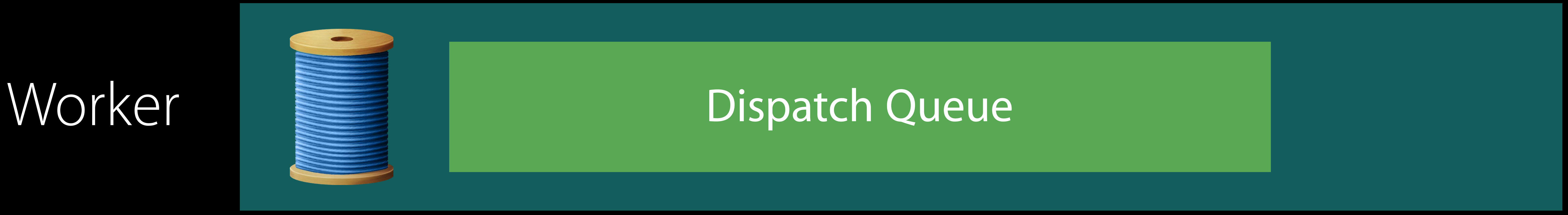
Dispatch Queue

() -> ()

Dispatch Queues and Run Loops



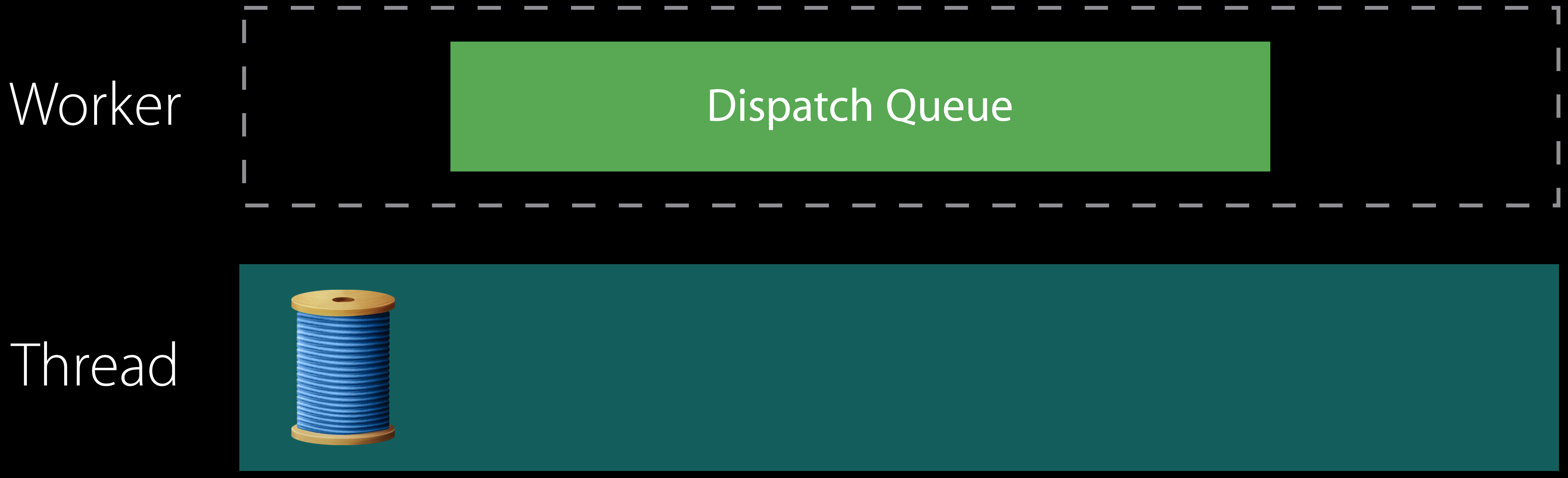
Dispatch Queues and Run Loops



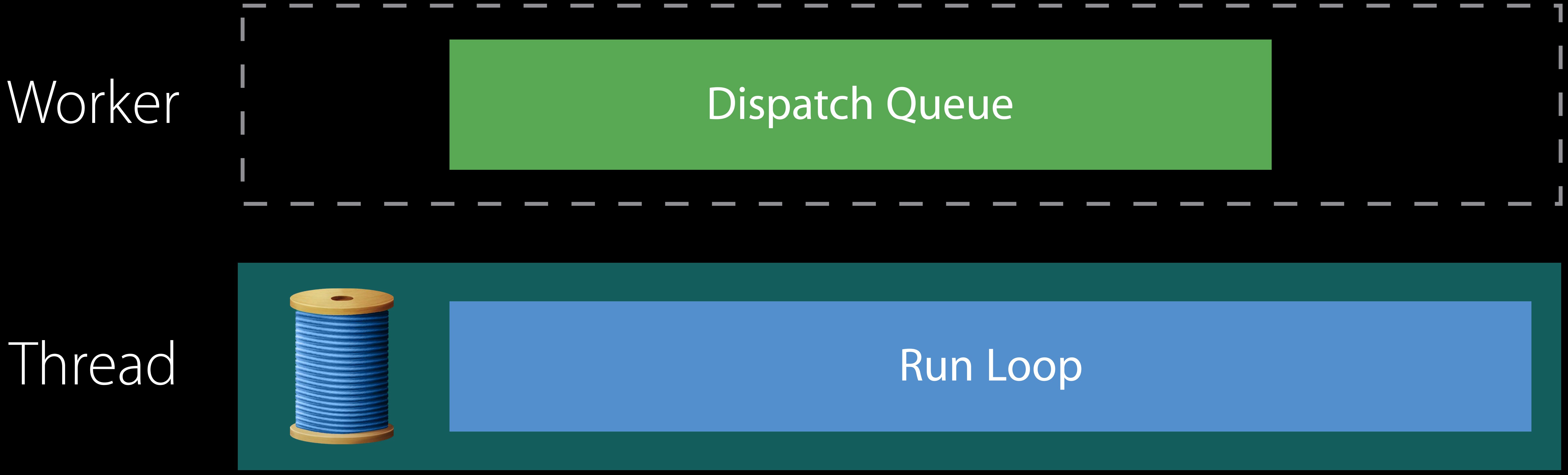
Dispatch Queues and Run Loops

Worker
Dispatch Queue

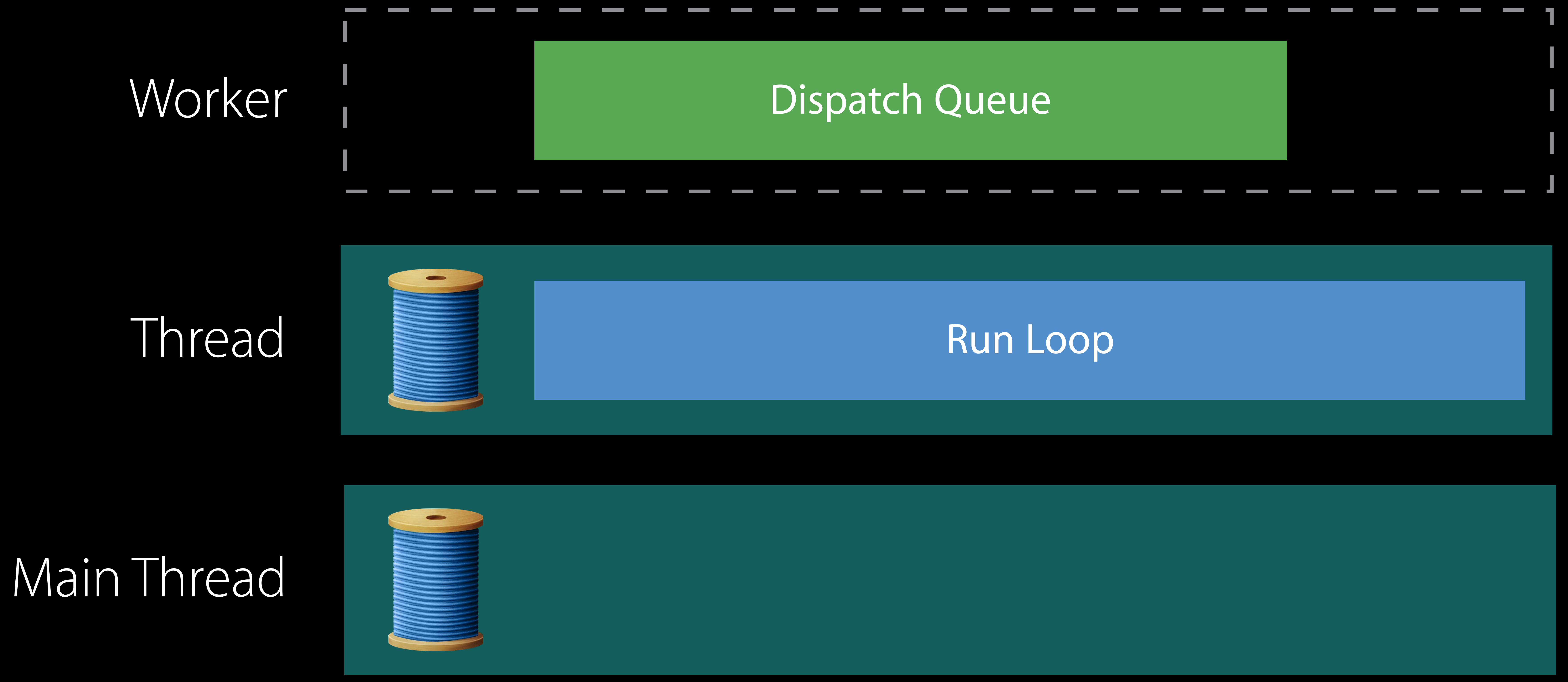
Dispatch Queues and Run Loops



Dispatch Queues and Run Loops



Dispatch Queues and Run Loops



Worker

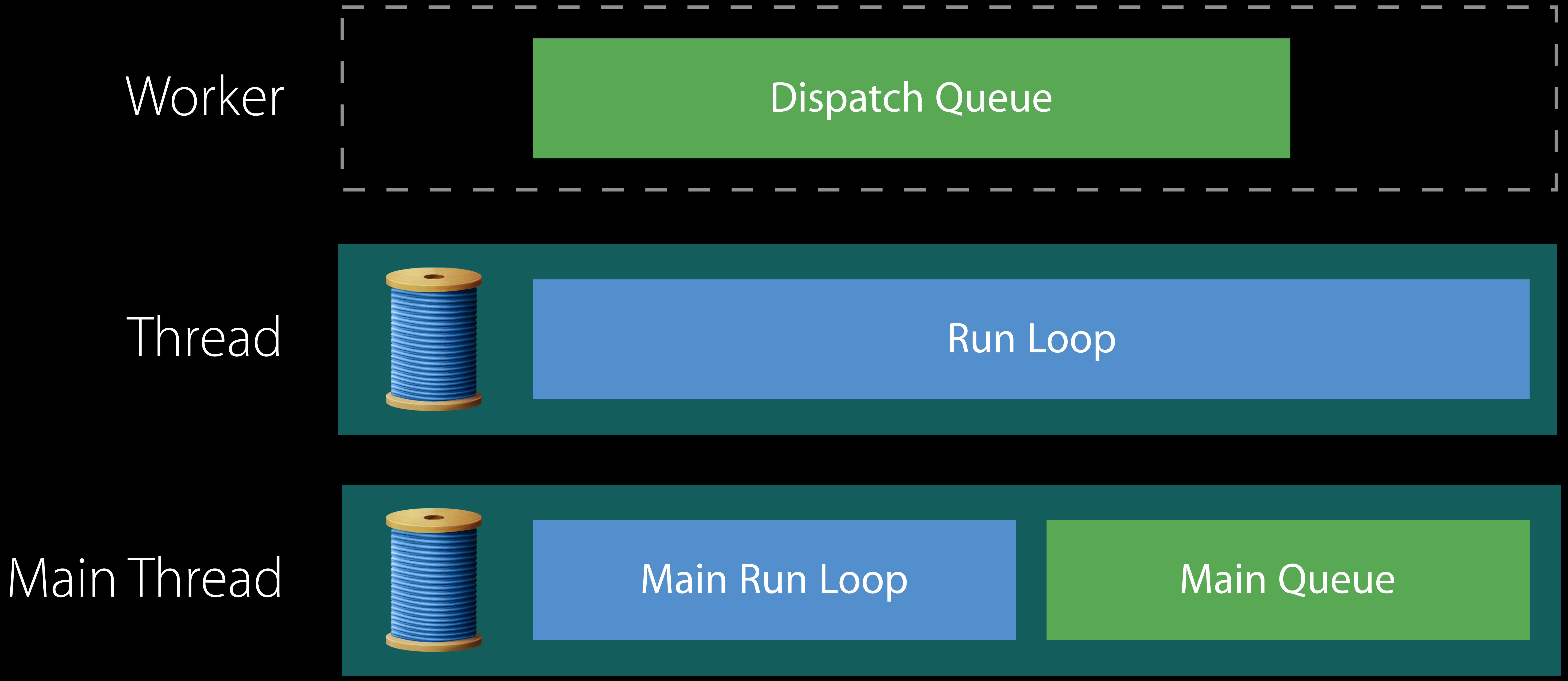
Dispatch Queue

Thread

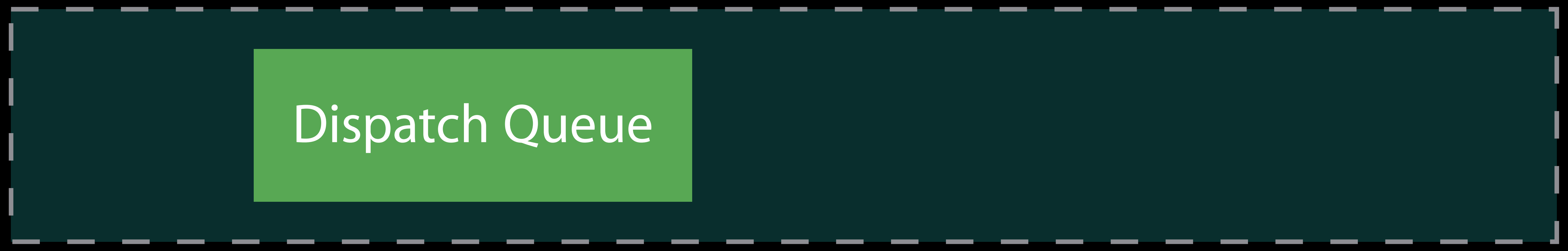
Run Loop

Main Thread

Dispatch Queues and Run Loops

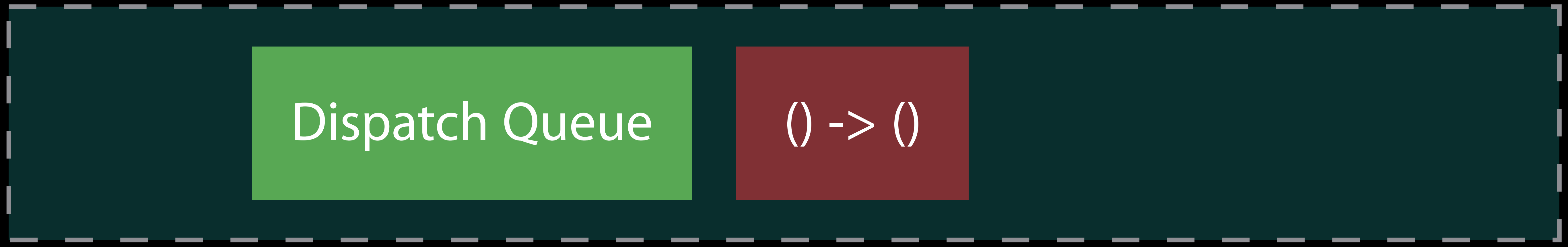


Asynchronous Execution



Dispatch Queue

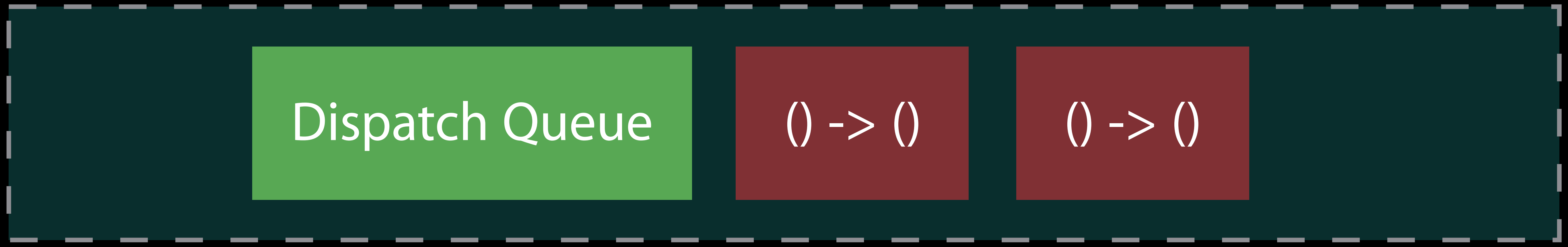
Asynchronous Execution



Dispatch Queue

() -> ()

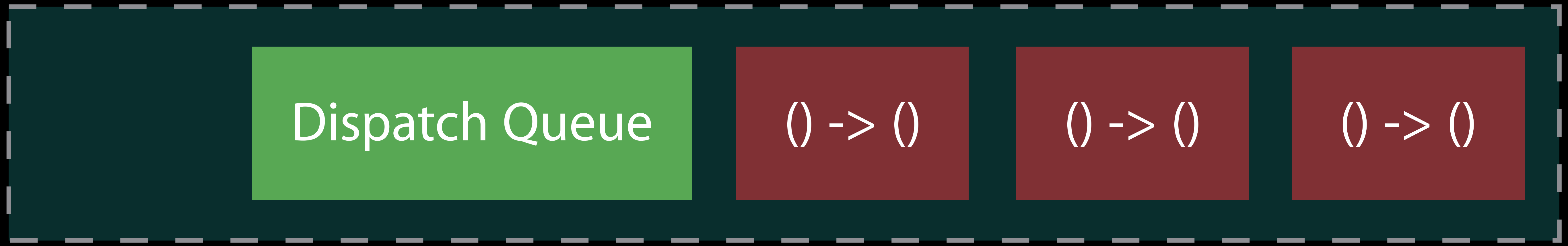
Asynchronous Execution


Dispatch Queue

() -> ()

() -> ()

Asynchronous Execution



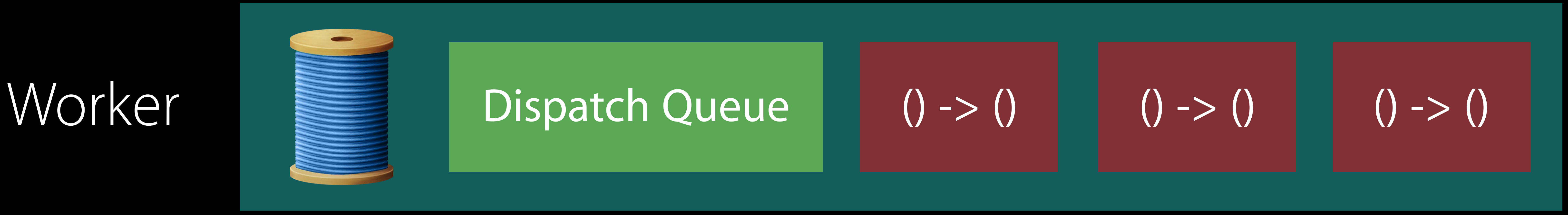
Dispatch Queue

() -> ()

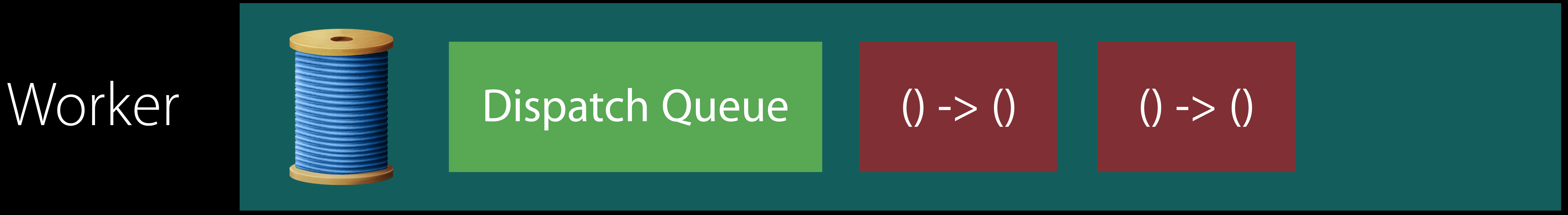
() -> ()

() -> ()

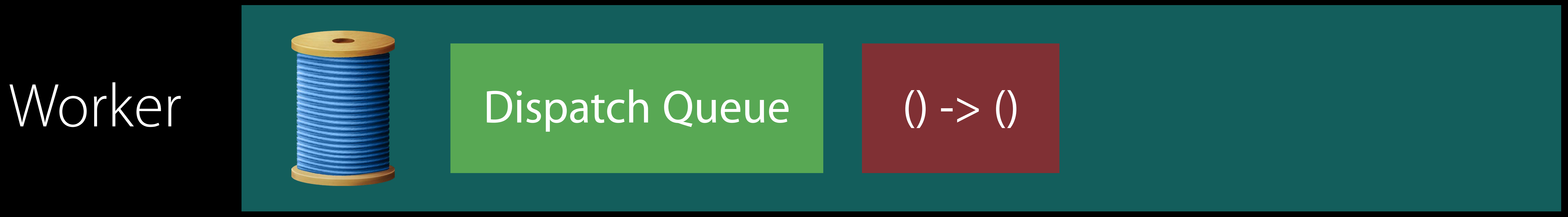
Asynchronous Execution



Asynchronous Execution



Asynchronous Execution



Worker

Dispatch Queue

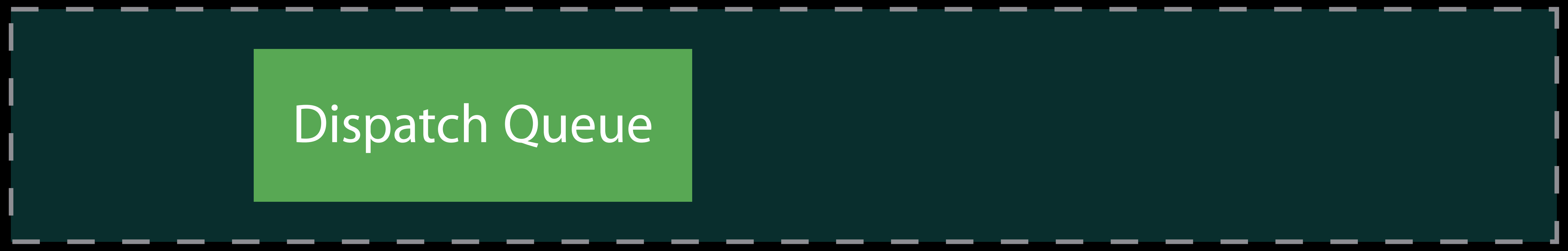
() -> ()

Asynchronous Execution

Worker

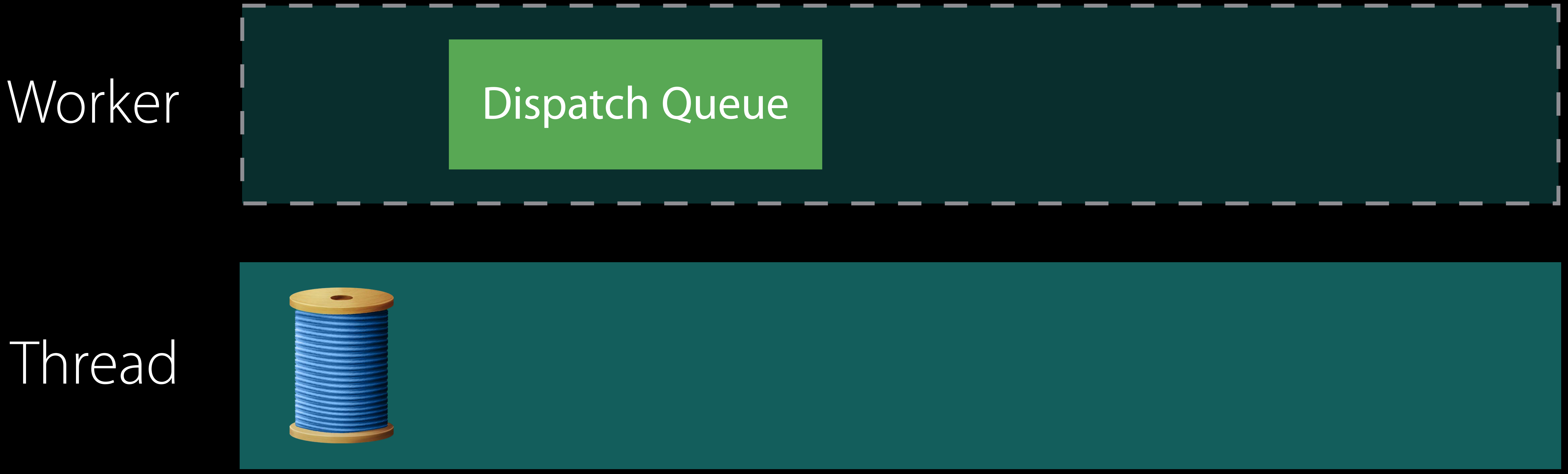
Dispatch Queue

Asynchronous Execution

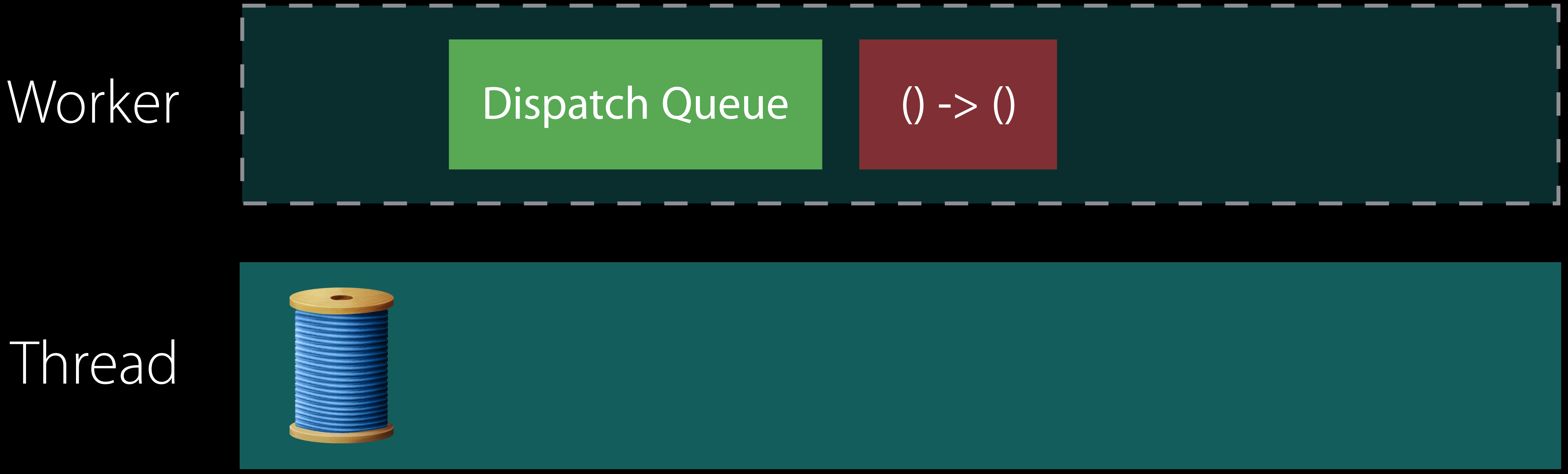


Dispatch Queue

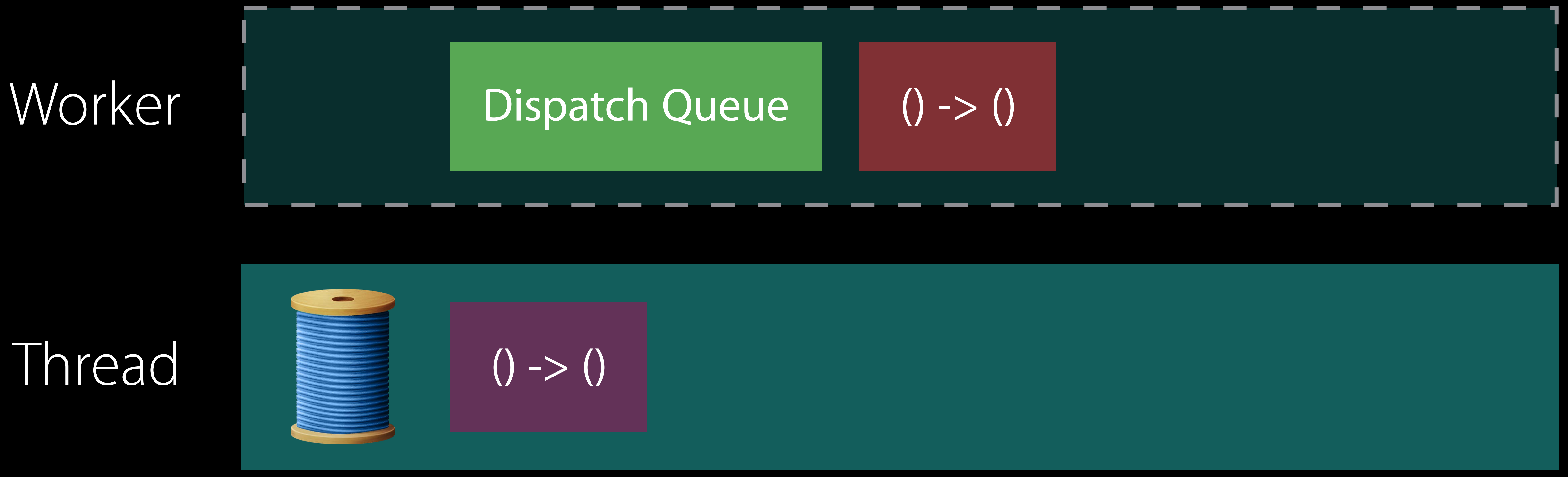
Synchronous Execution



Synchronous Execution



Synchronous Execution



Worker

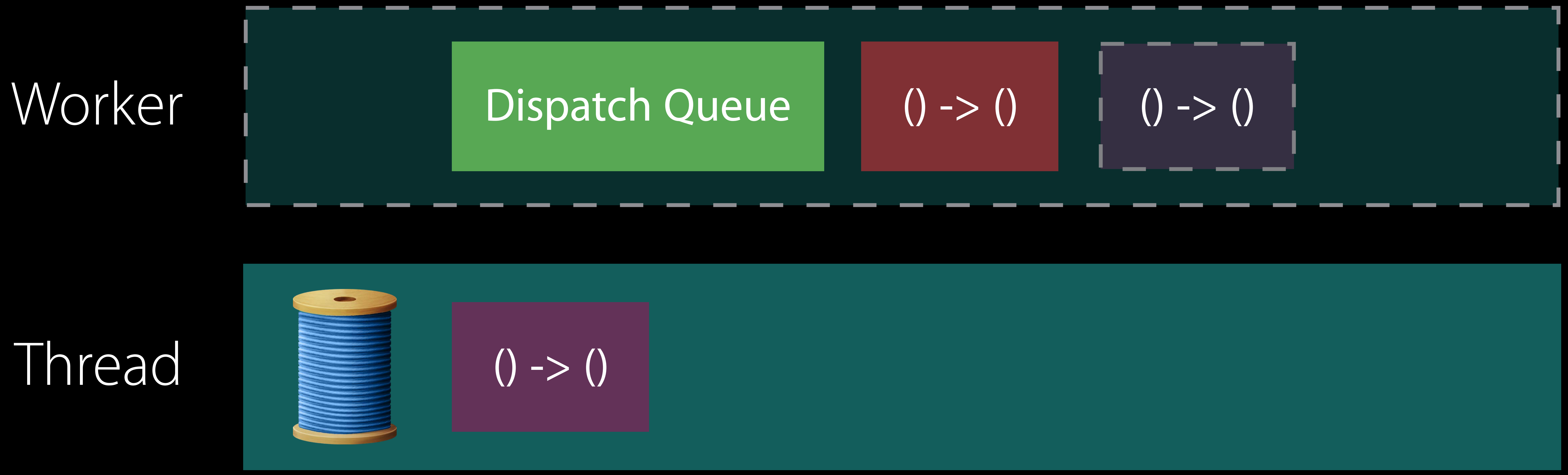
Dispatch Queue

() -> ()

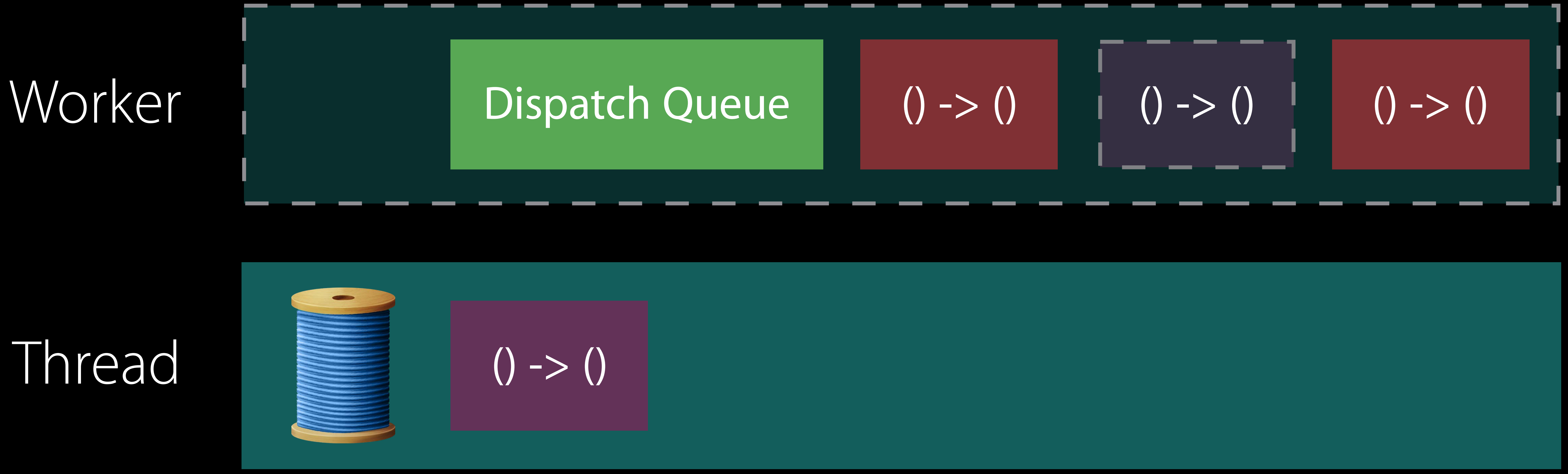
Thread

() -> ()

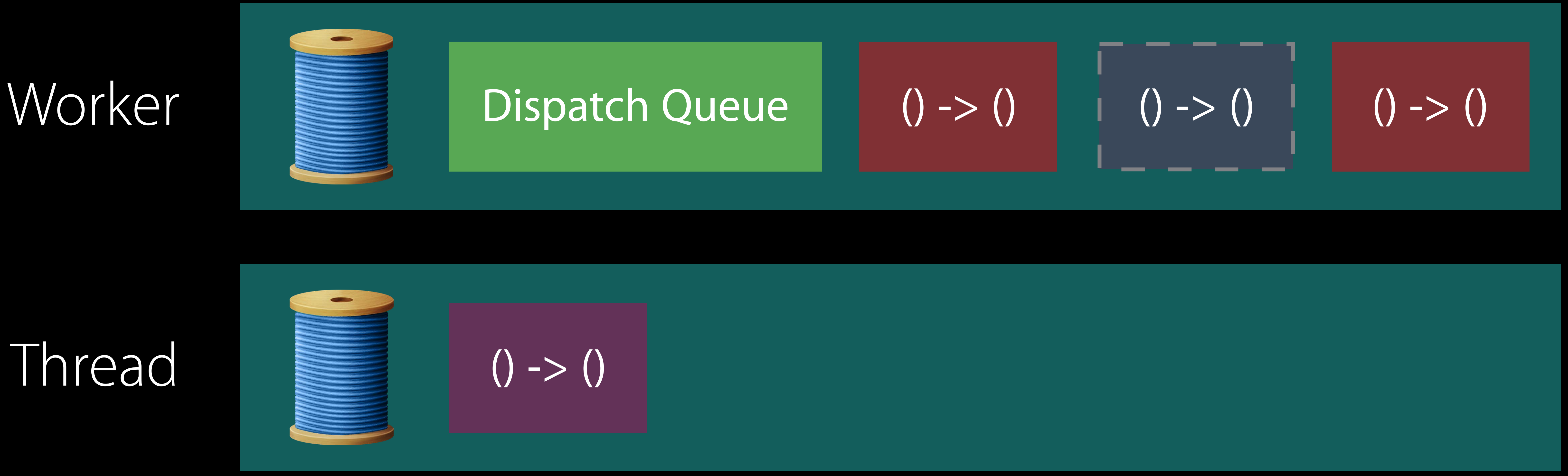
Synchronous Execution



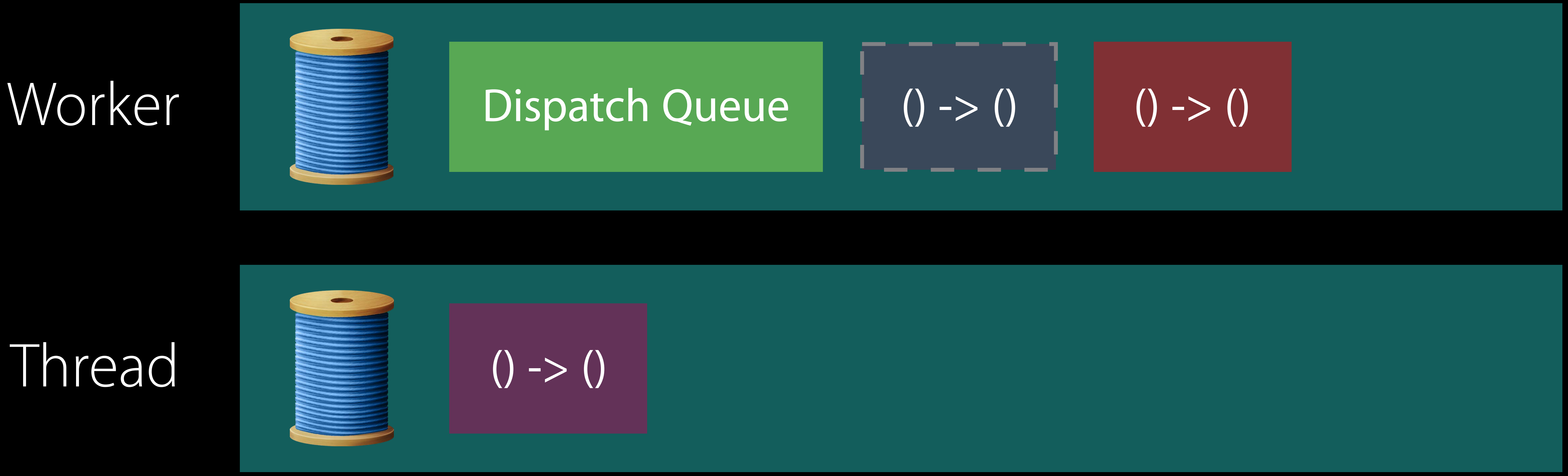
Synchronous Execution



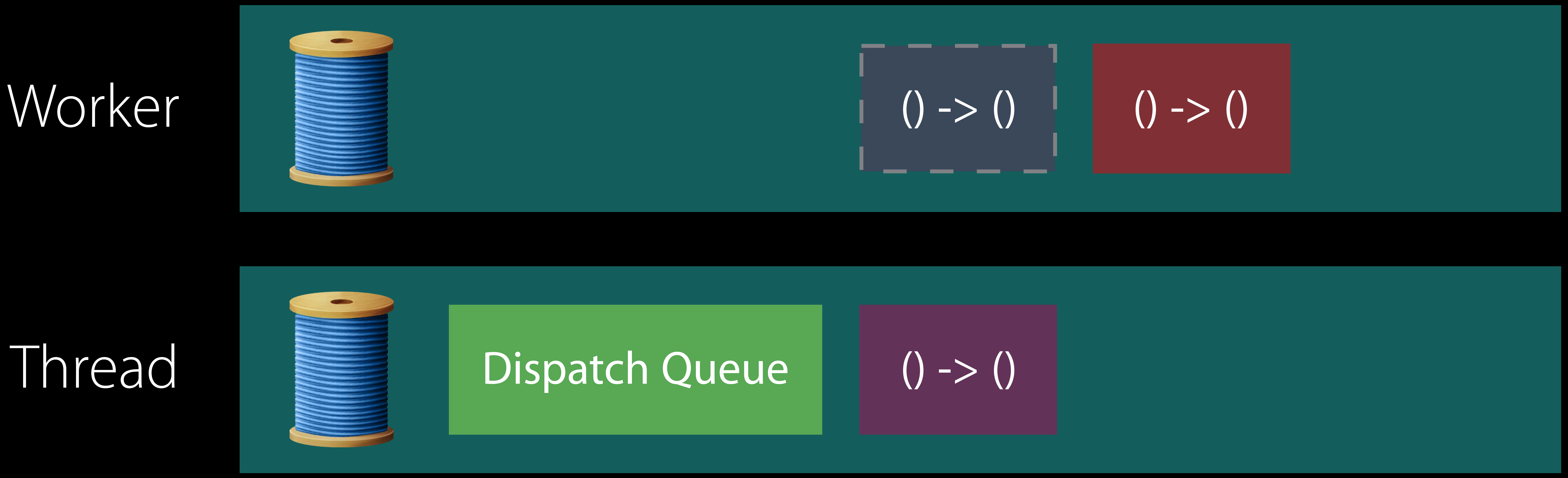
Synchronous Execution



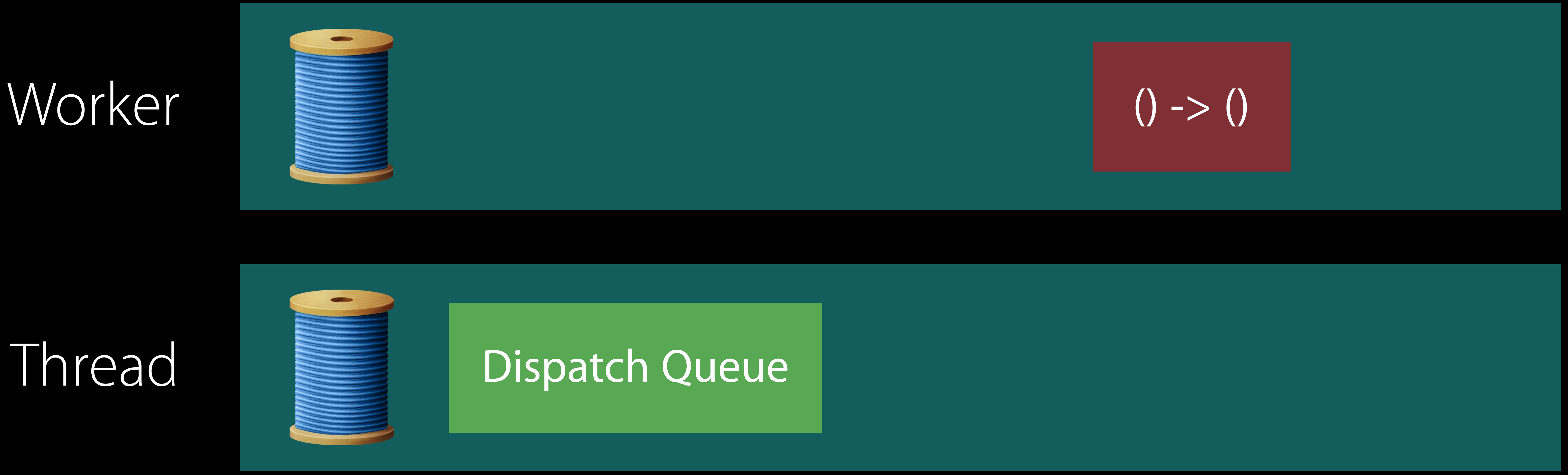
Synchronous Execution



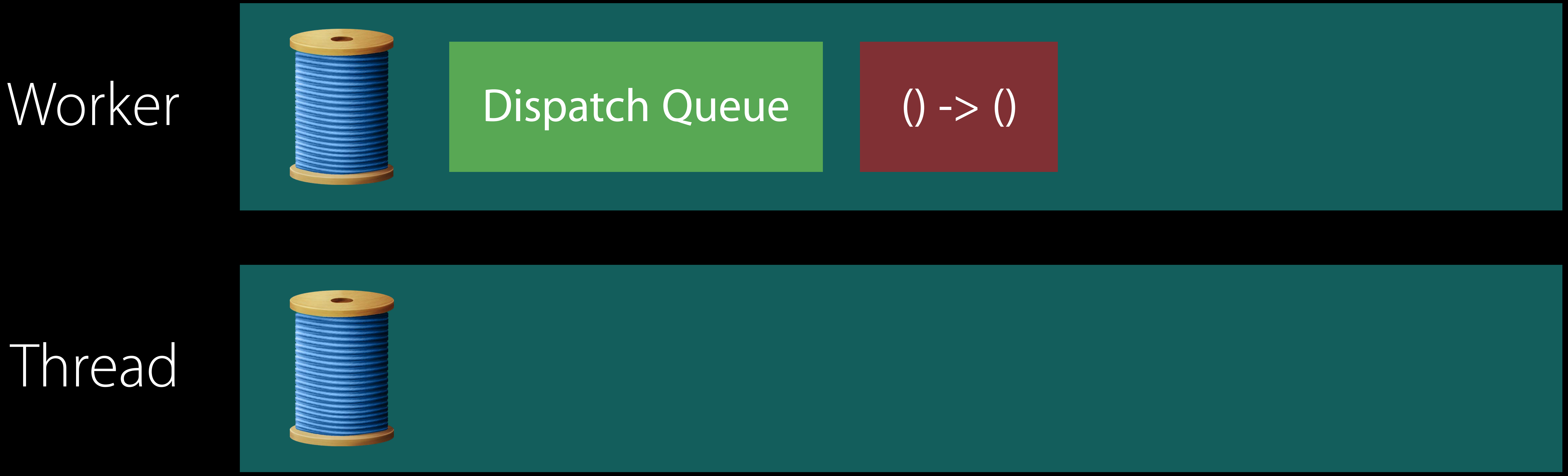
Synchronous Execution



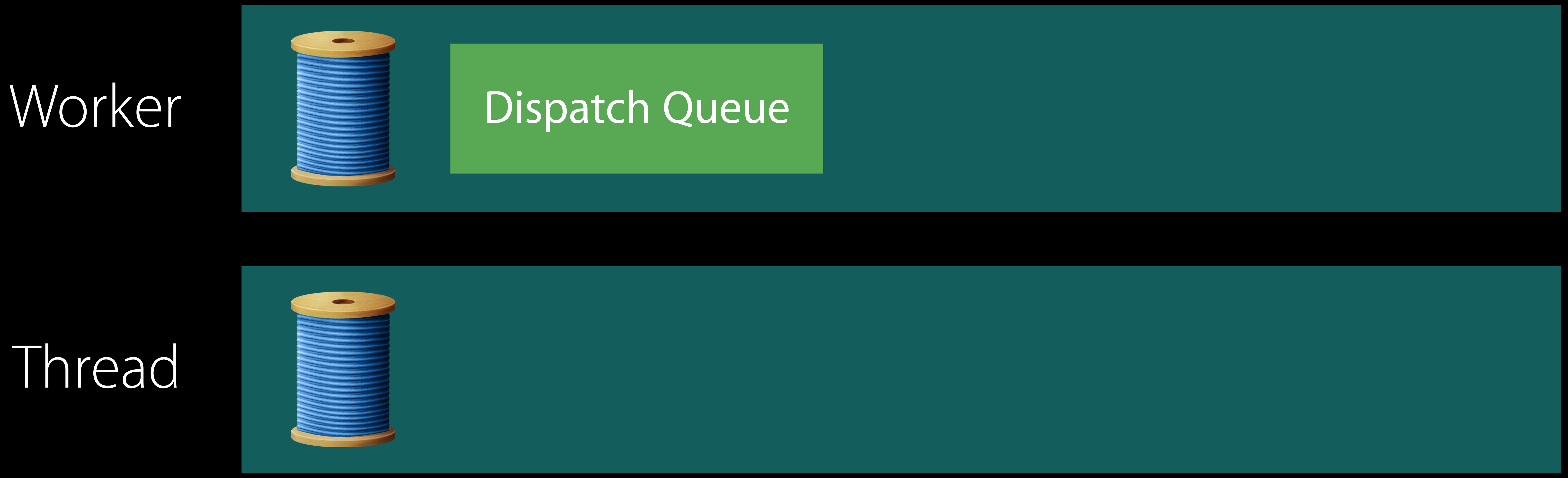
Synchronous Execution



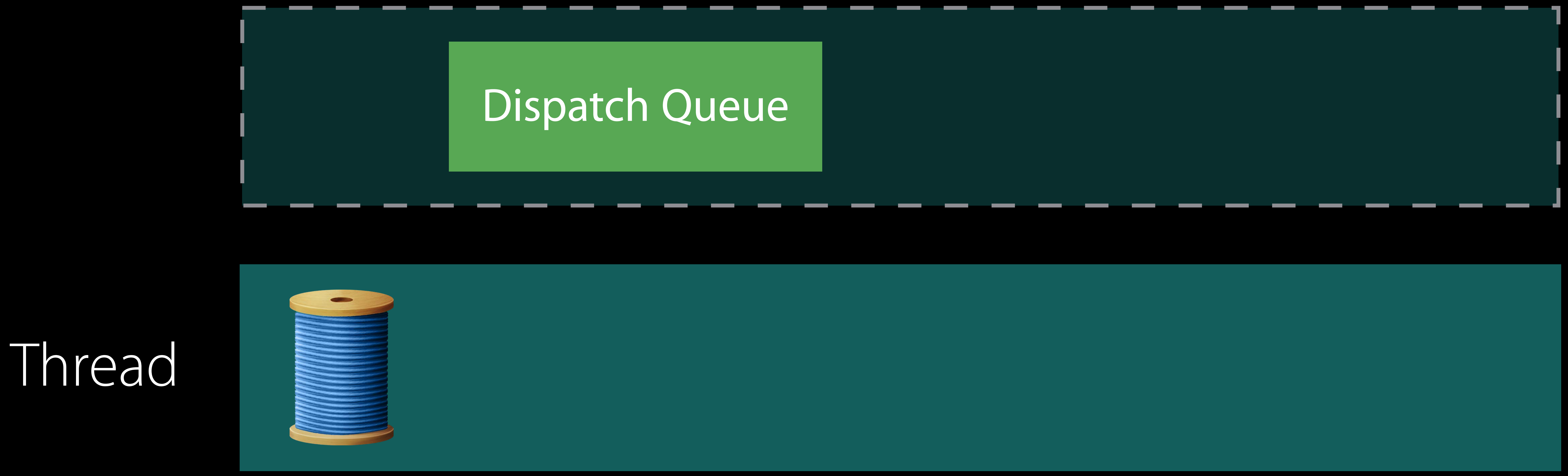
Synchronous Execution



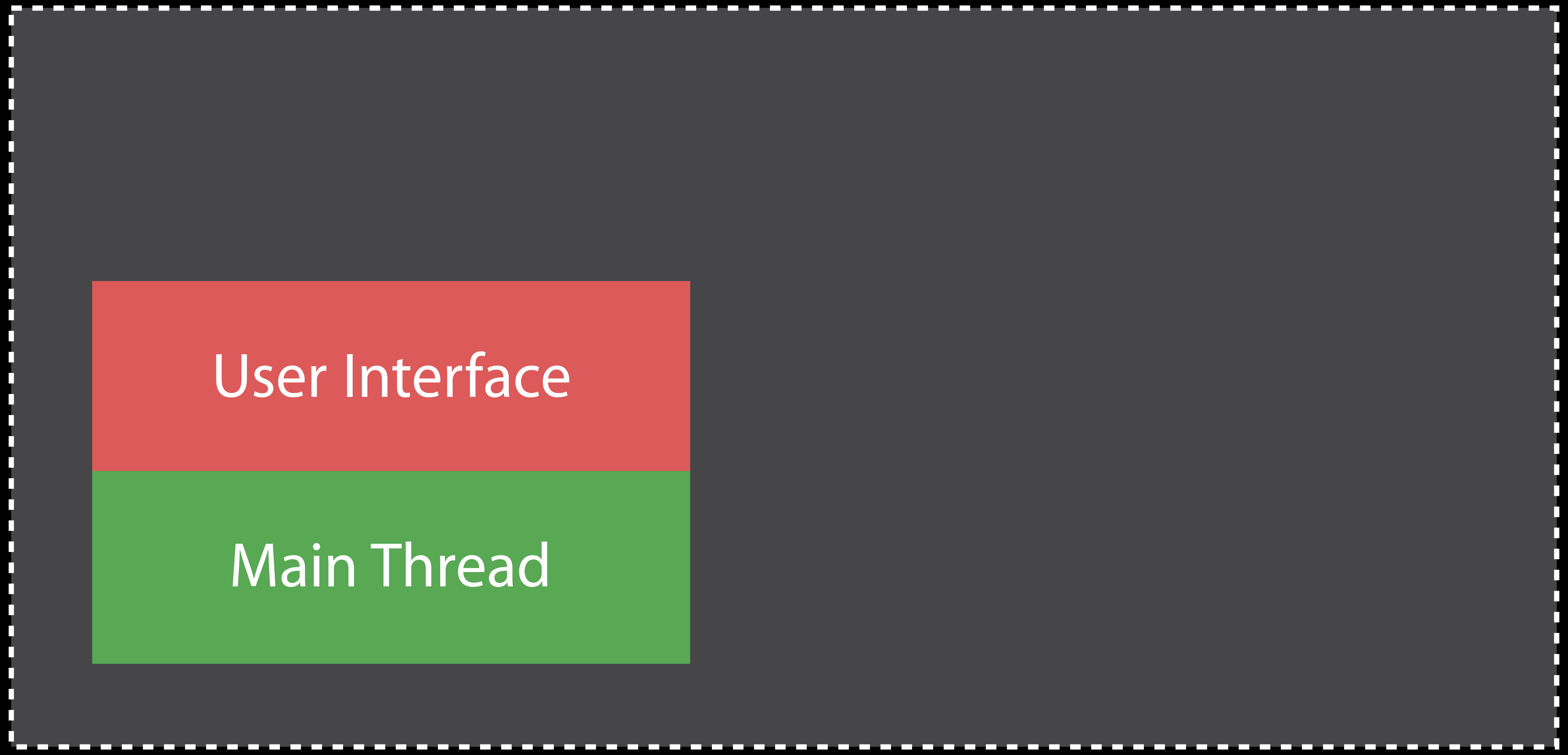
Synchronous Execution



Synchronous Execution



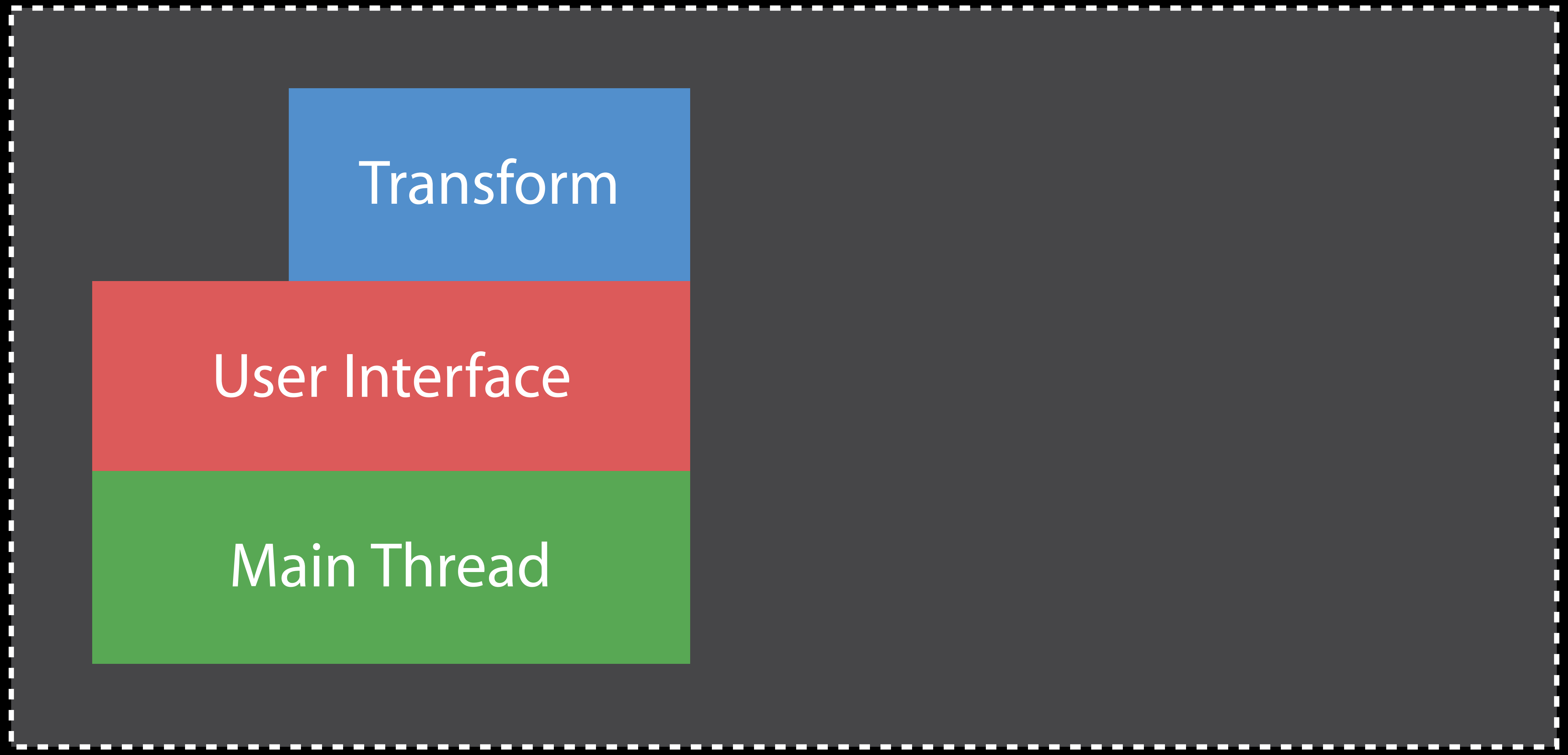
Getting Work Off Your Main Thread



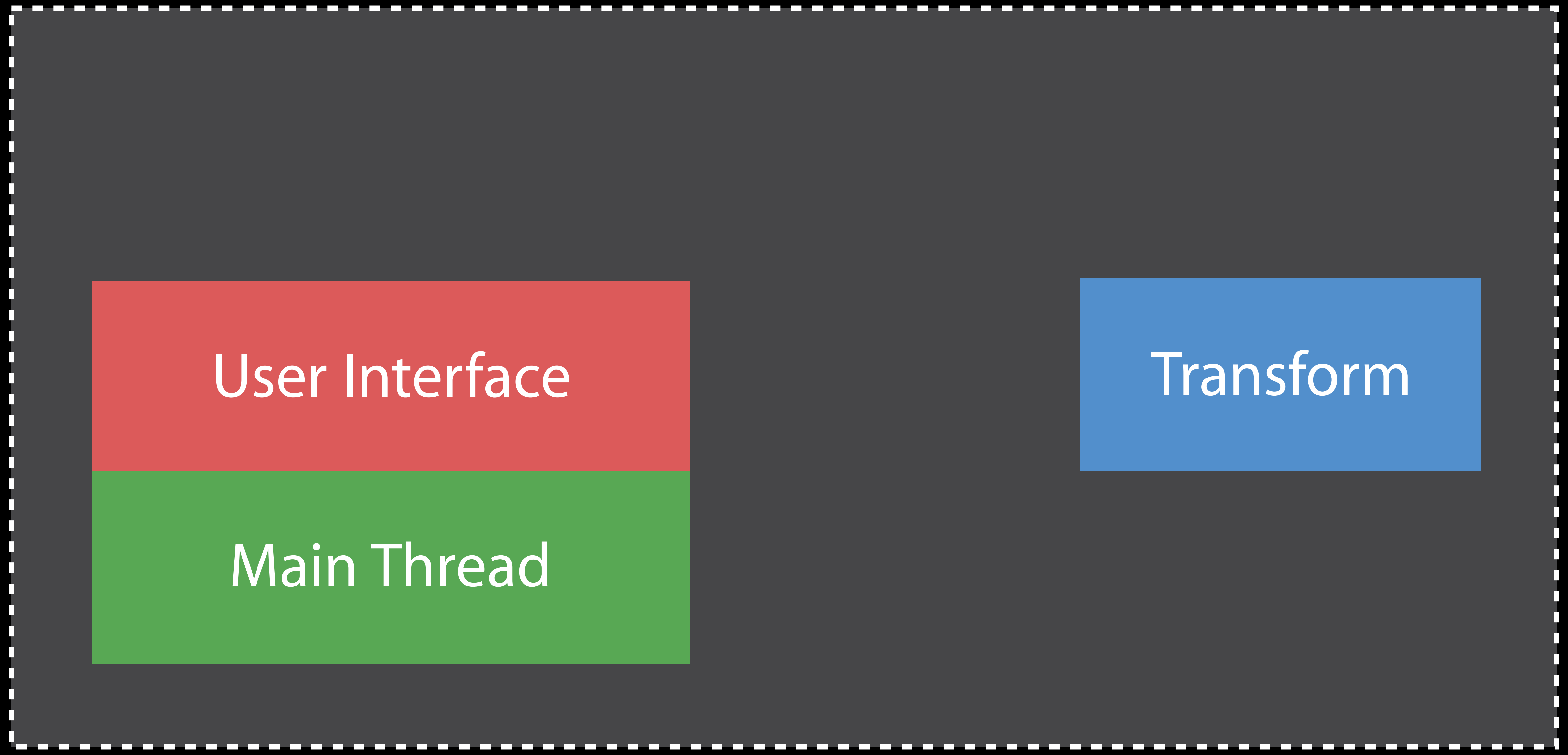
User Interface

Main Thread

Getting Work Off Your Main Thread



Getting Work Off Your Main Thread

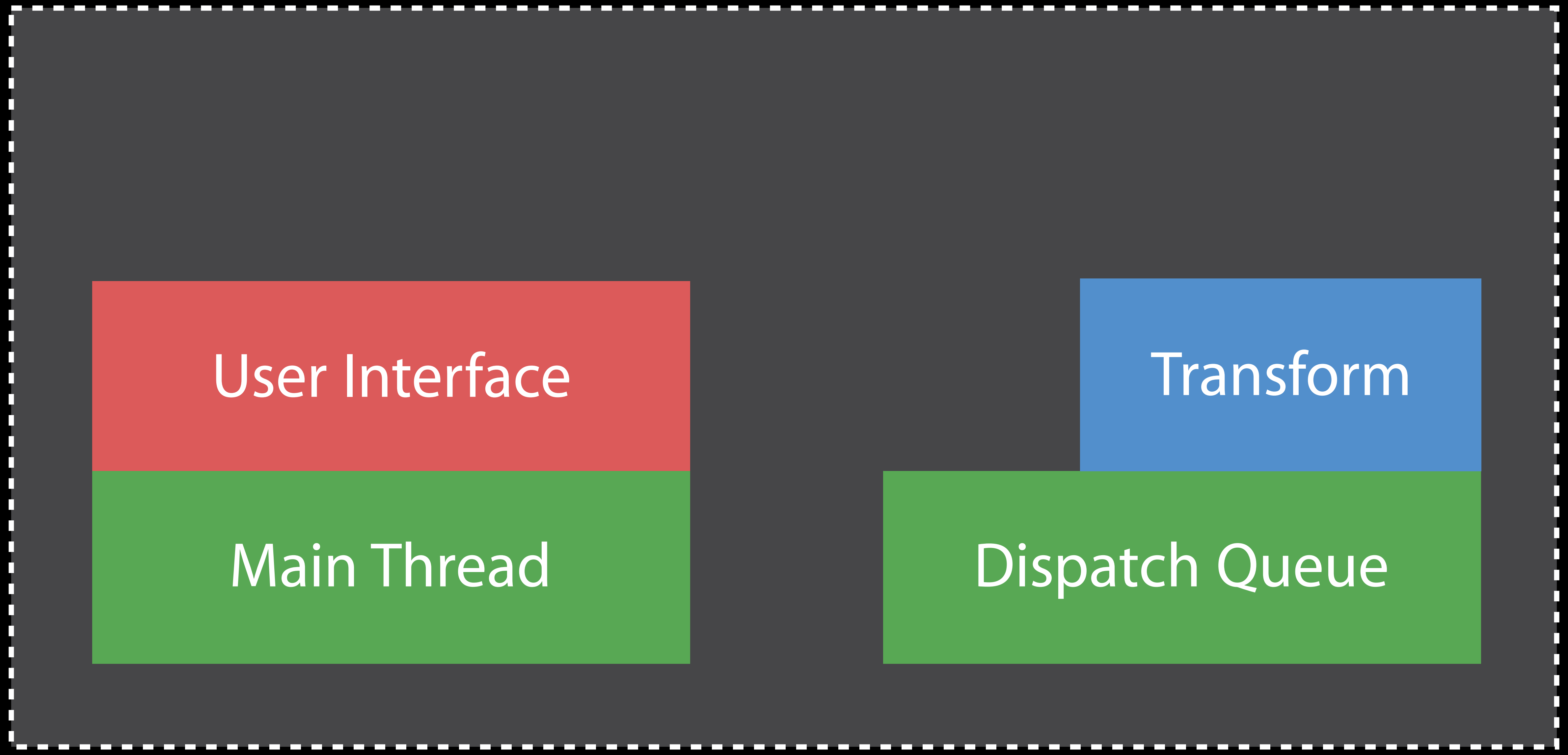


User Interface

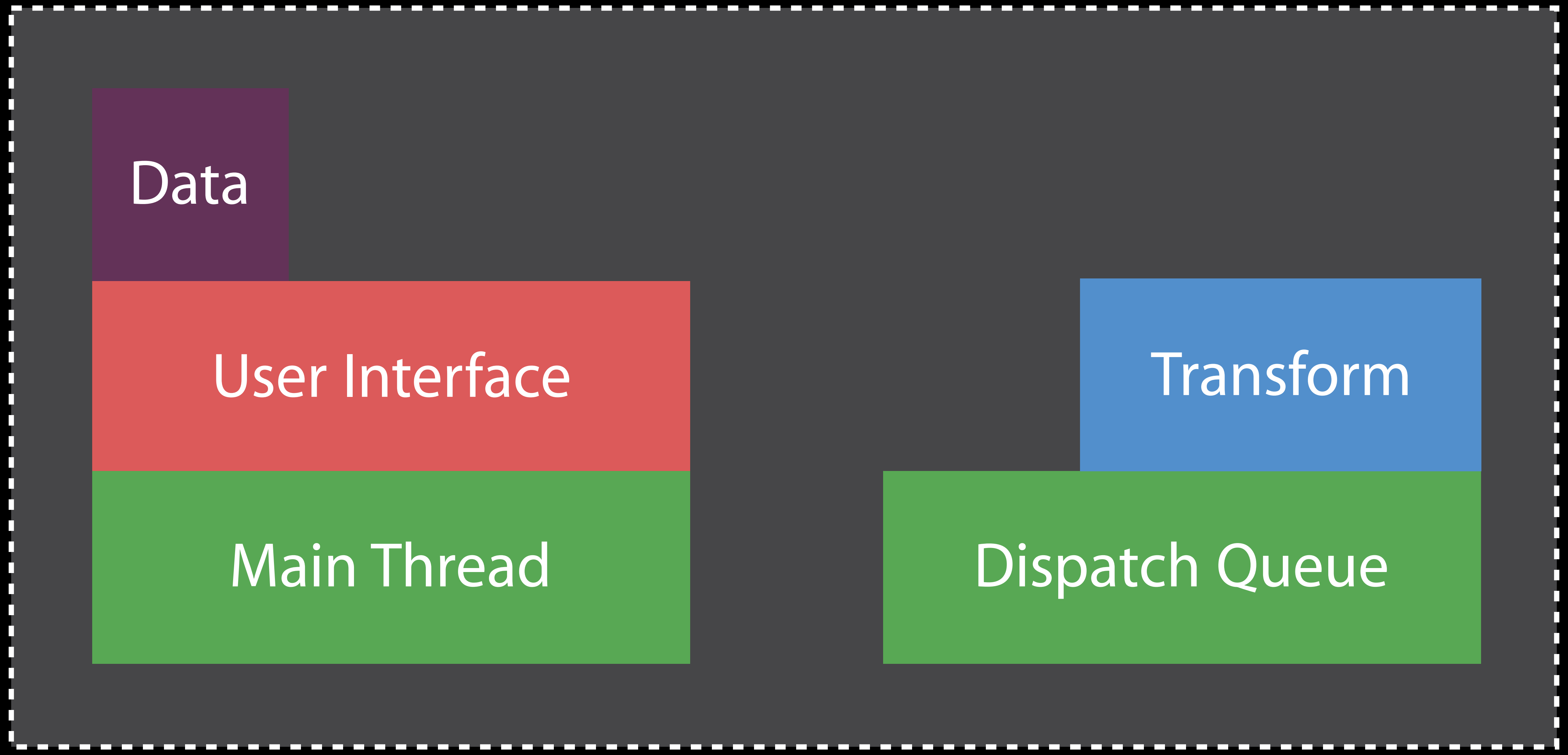
Main Thread

Transform

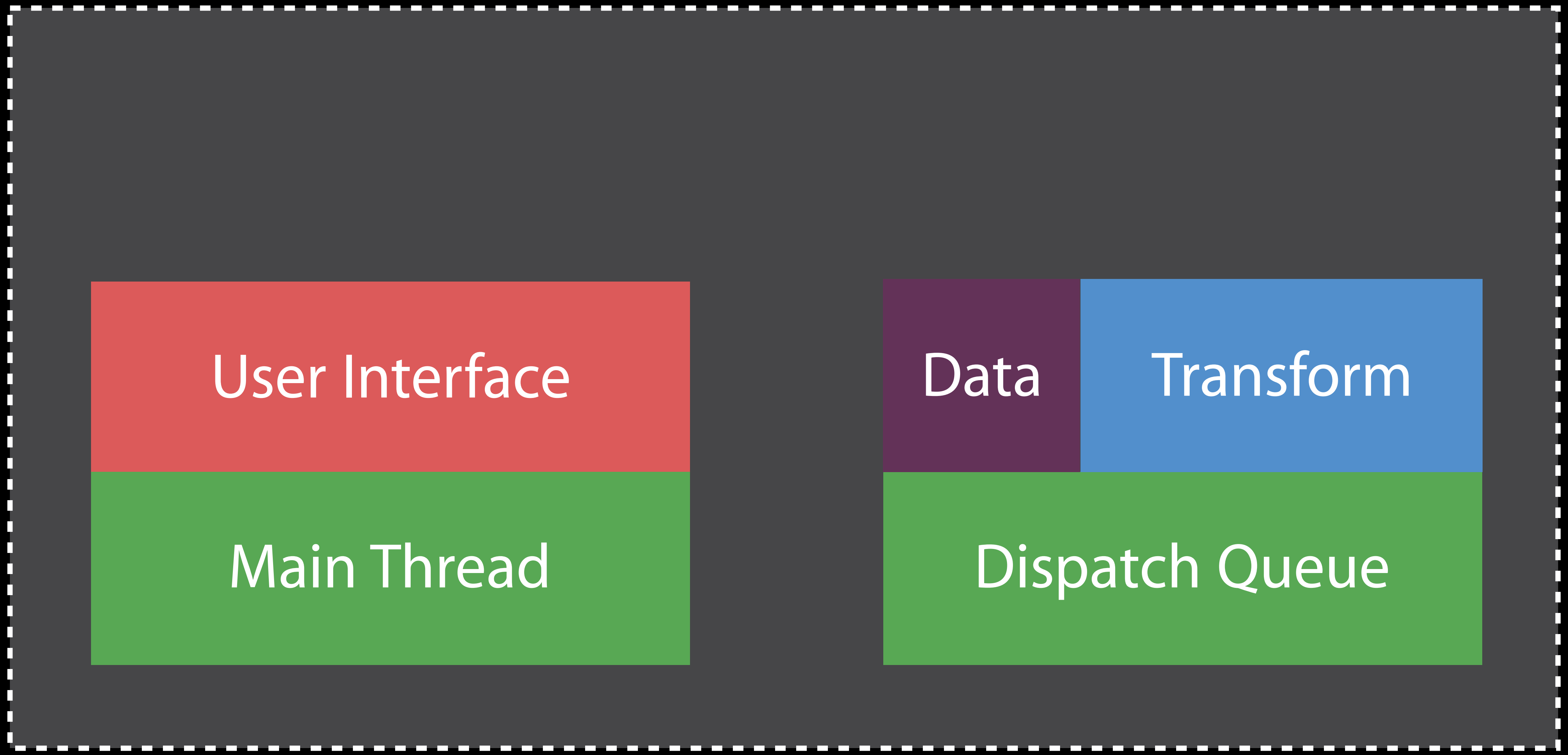
Getting Work Off Your Main Thread



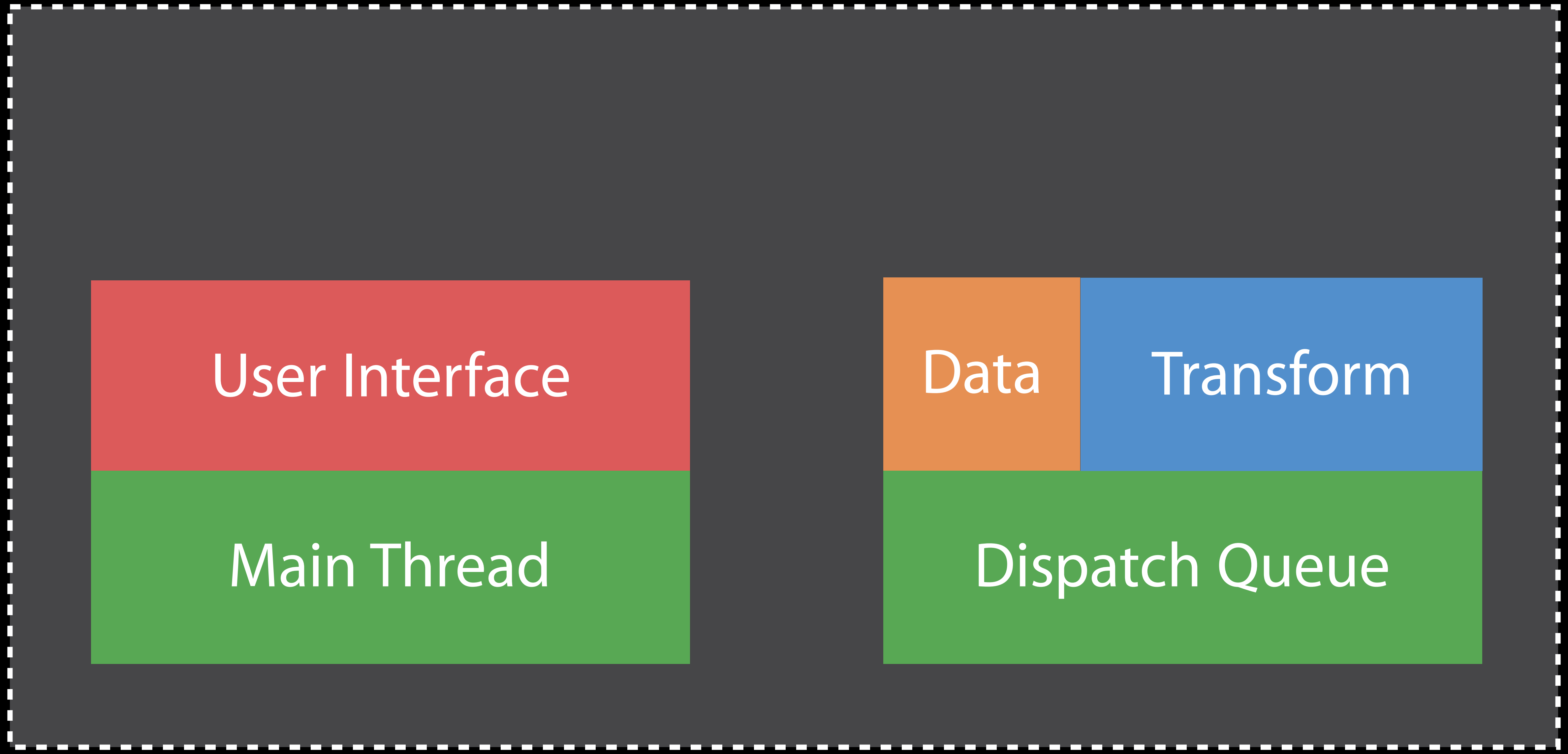
Getting Work Off Your Main Thread



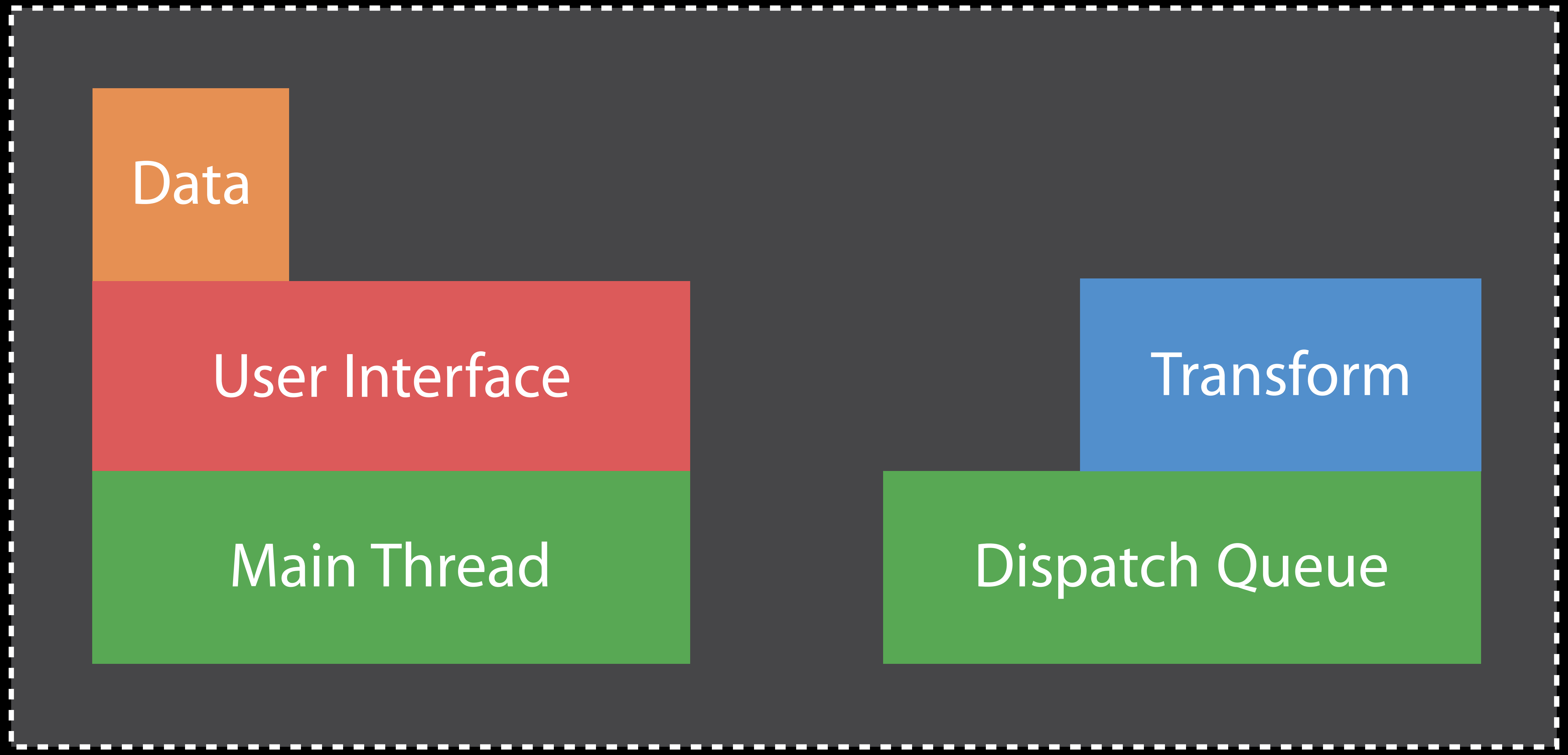
Getting Work Off Your Main Thread



Getting Work Off Your Main Thread



Getting Work Off Your Main Thread



Getting Work Off Your Main Thread

Create a Dispatch Queue to which you submit work

```swift
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)




}
```

Getting Work Off Your Main Thread

Create a Dispatch Queue to which you submit work

```swift
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)




}
```

Getting Work Off Your Main Thread

Create a Dispatch Queue to which you submit work

Dispatch Queues execute work items in FIFO order

```swift
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)




}
```

Getting Work Off Your Main Thread

Create a Dispatch Queue to which you submit work

Dispatch Queues execute work items in FIFO order

Use `.async` to execute your work on the queue

```
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)




}
```

Getting Back to Your Main Thread

Dispatch main queue executes all items on the main thread

```swift
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)

    DispatchQueue.main.async {
      imageView.image = smallImage
    }
}
```

Getting Back to Your Main Thread

Dispatch main queue executes all items on the main thread

```swift
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)

    DispatchQueue.main.async {
      imageView.image = smallImage
    }
}
```

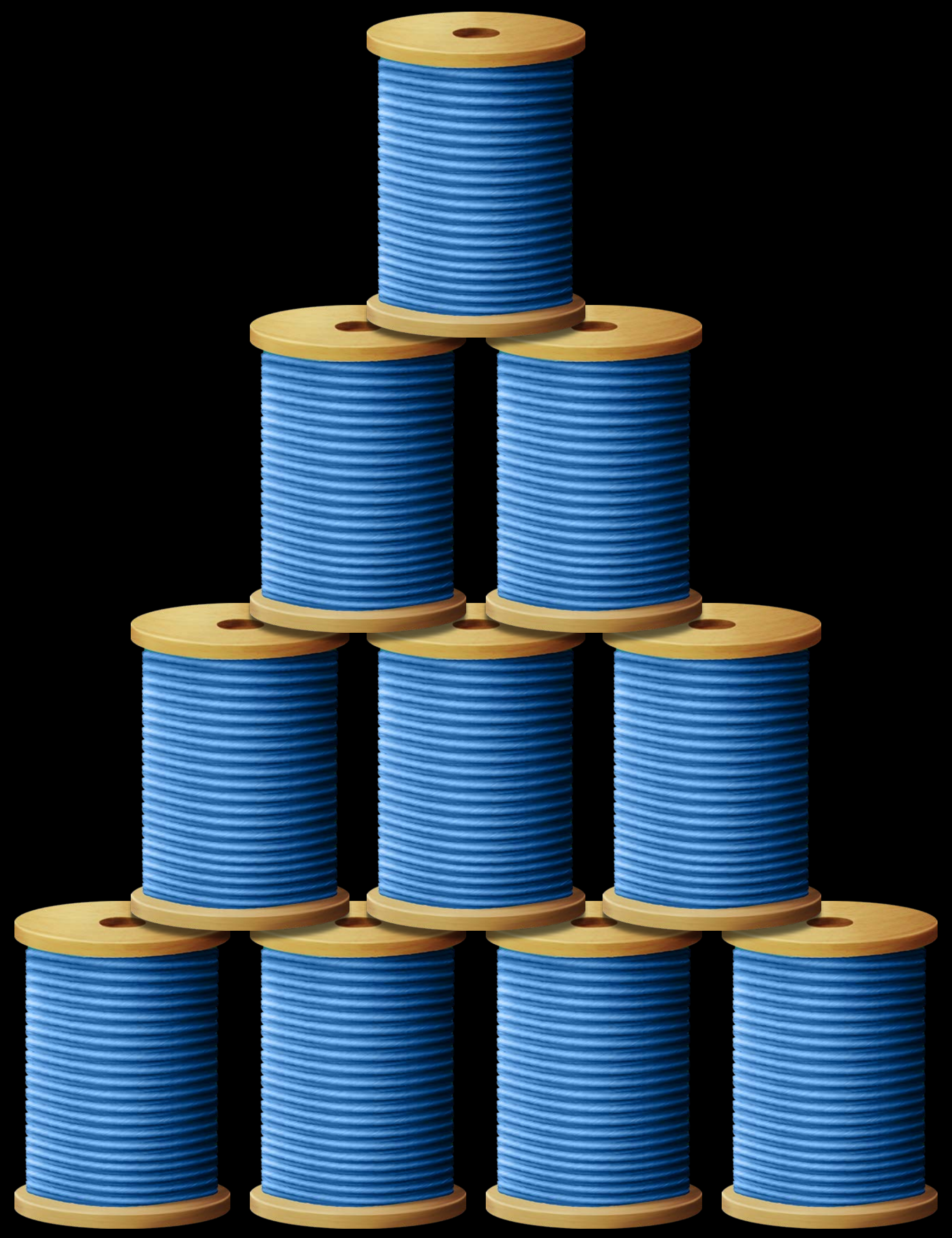
Getting Back to Your Main Thread

Dispatch main queue executes all items on the main thread

Simple to chain work between queues

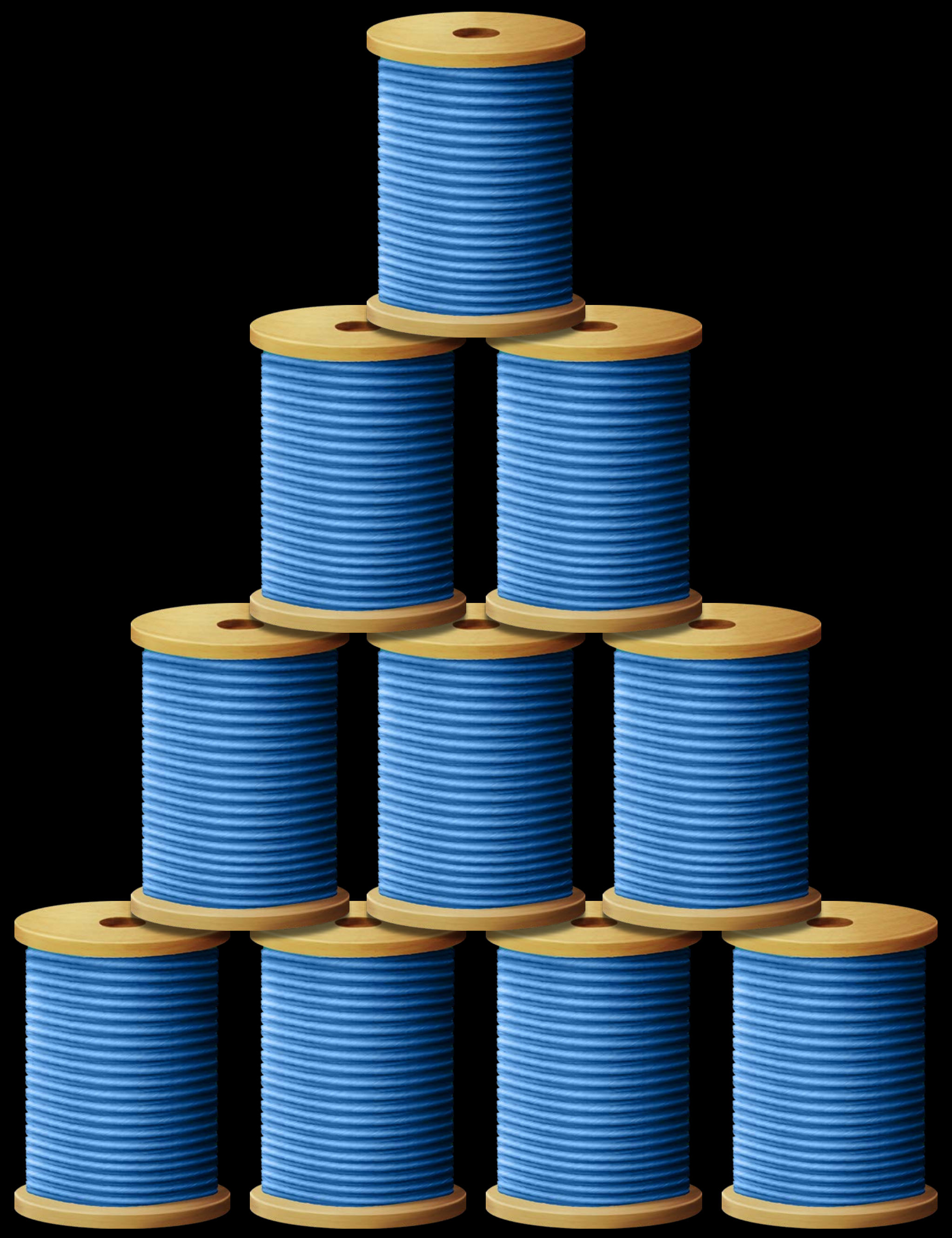
```swift
let queue = DispatchQueue(label: "com.example.imagetransform")

queue.async {
    let smallImage = image.resize(to: rect)

    DispatchQueue.main.async {
      imageView.image = smallImage
    }
}
```

Controlling Concurrency



Controlling Concurrency

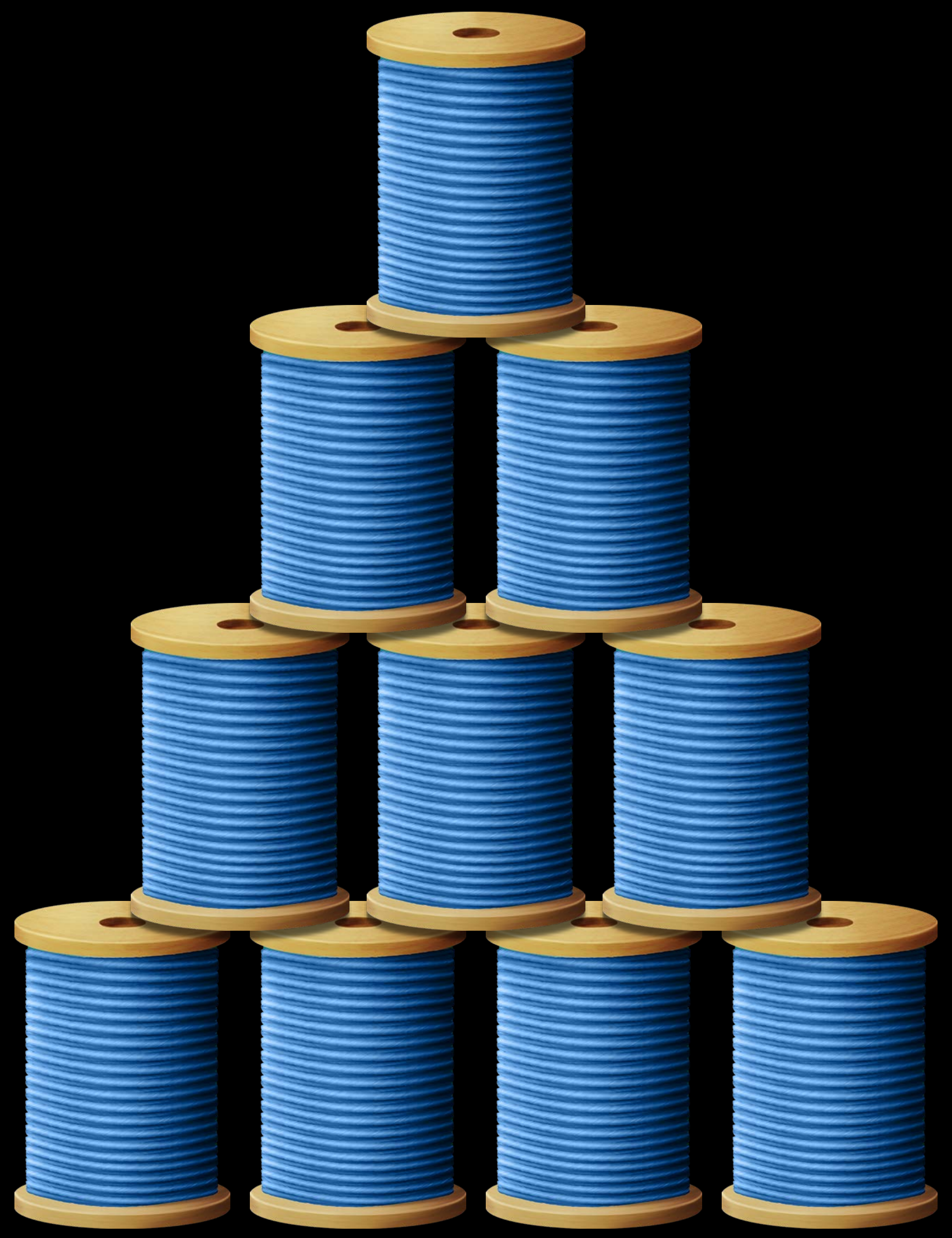
Thread pool will limit concurrency



Controlling Concurrency

Thread pool will limit concurrency

Worker threads that block can cause more to spawn

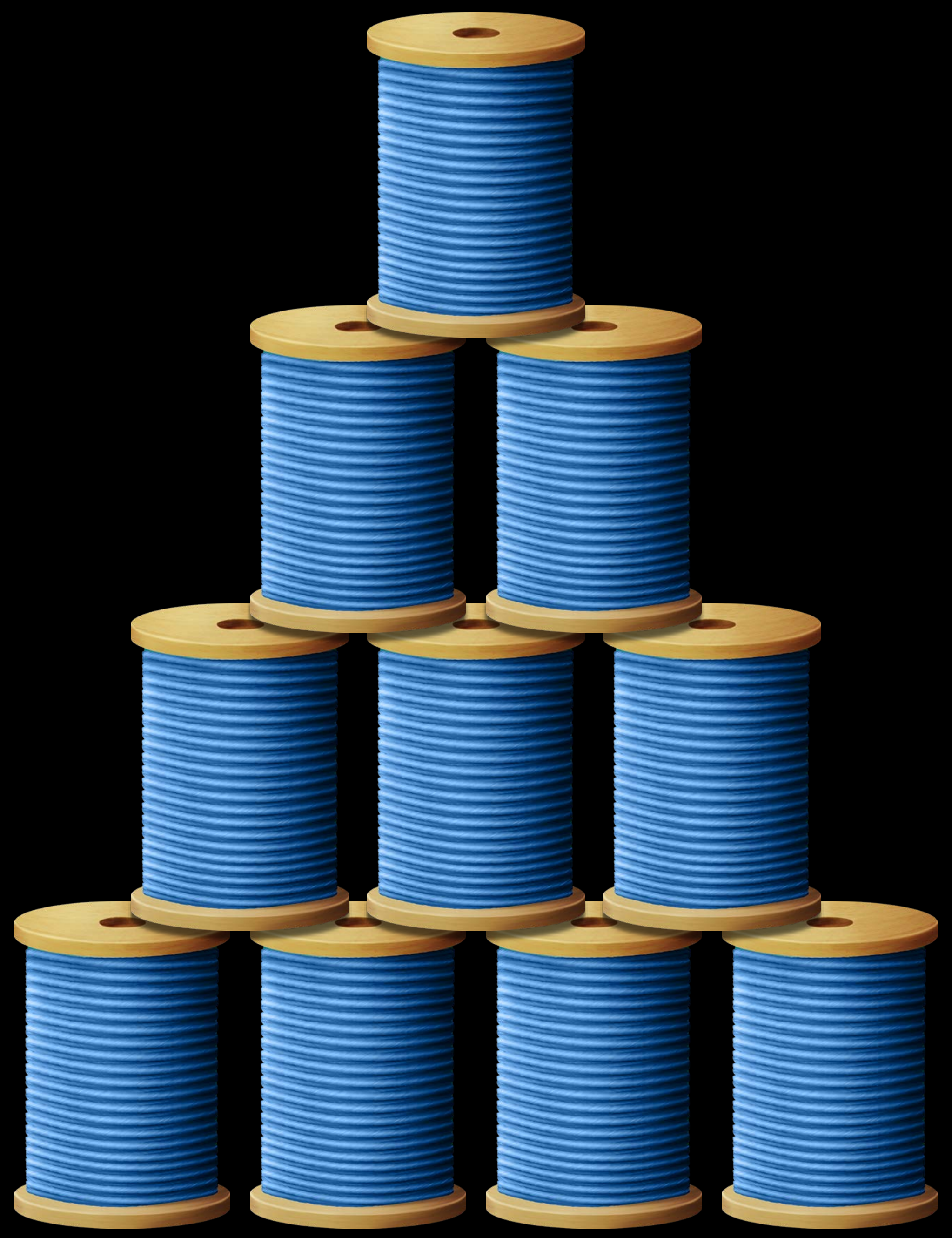


Controlling Concurrency

Thread pool will limit concurrency

Worker threads that block can cause more to spawn

Choosing the right number of queues to use is important

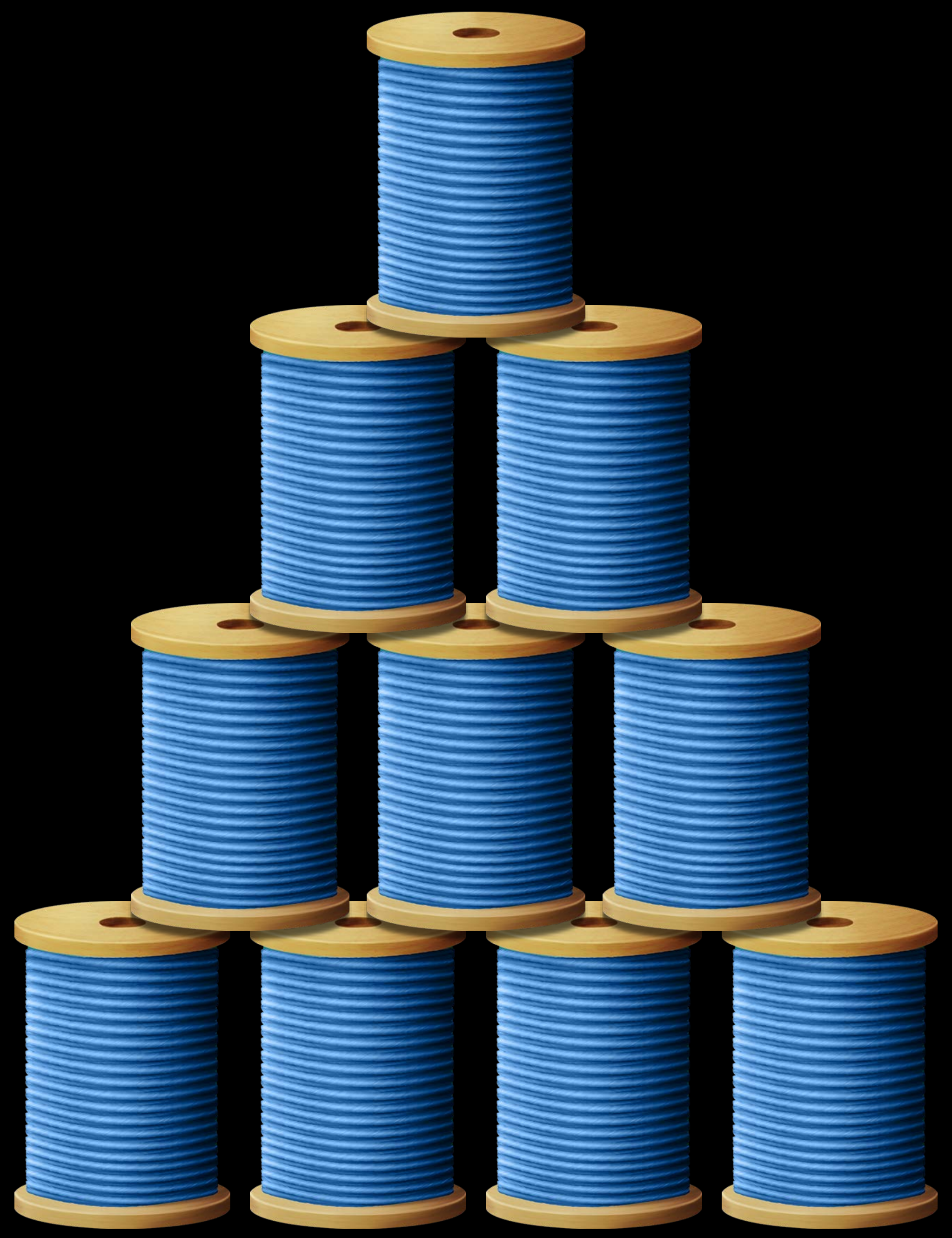


Controlling Concurrency

Thread pool will limit concurrency

Worker threads that block can cause more to spawn

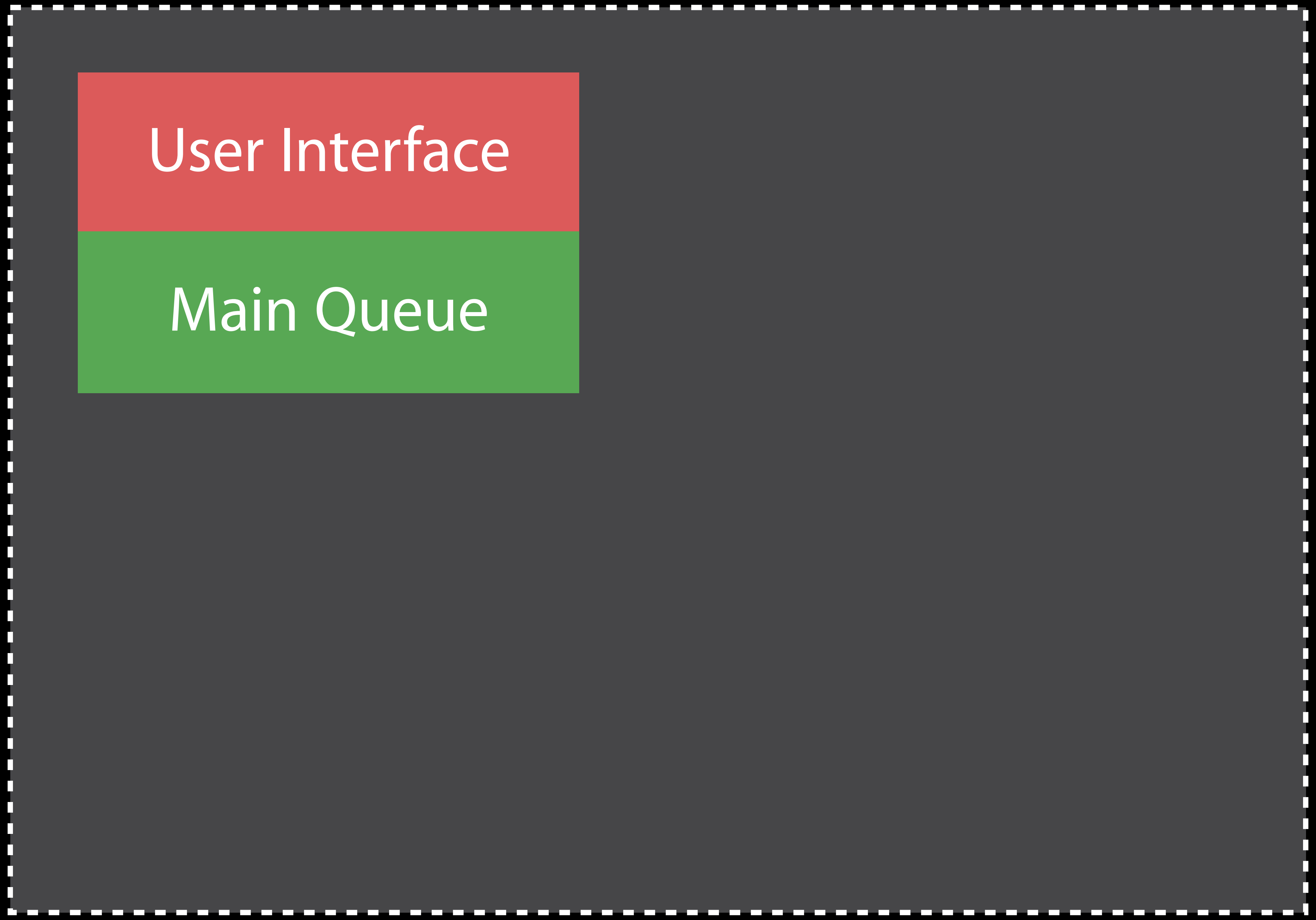
Choosing the right number of queues to use is important



Structuring Your Application

Structuring Your Application

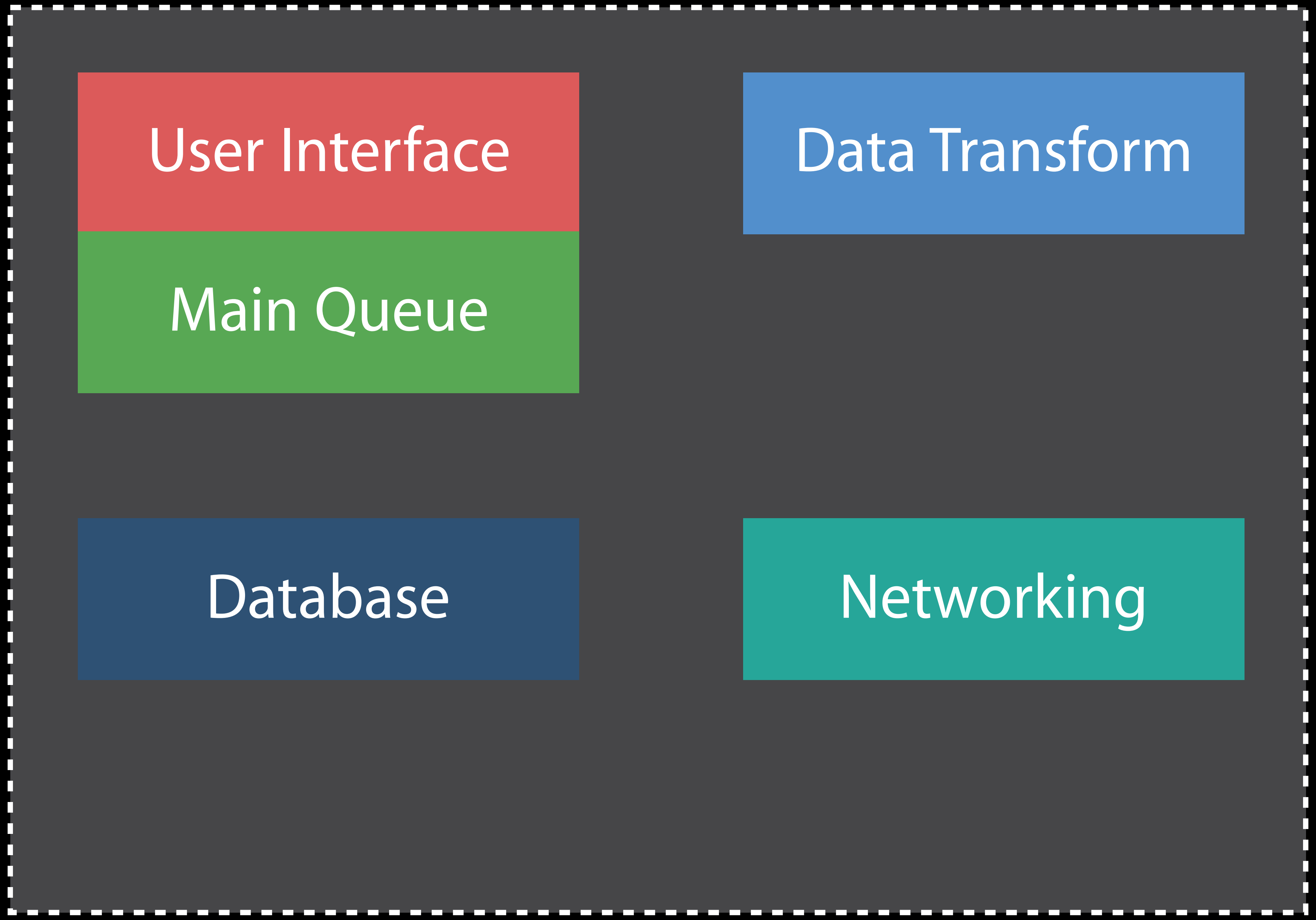
Identify areas of data flow in your application



Structuring Your Application

Identify areas of data flow in your application

Split into distinct subsystems

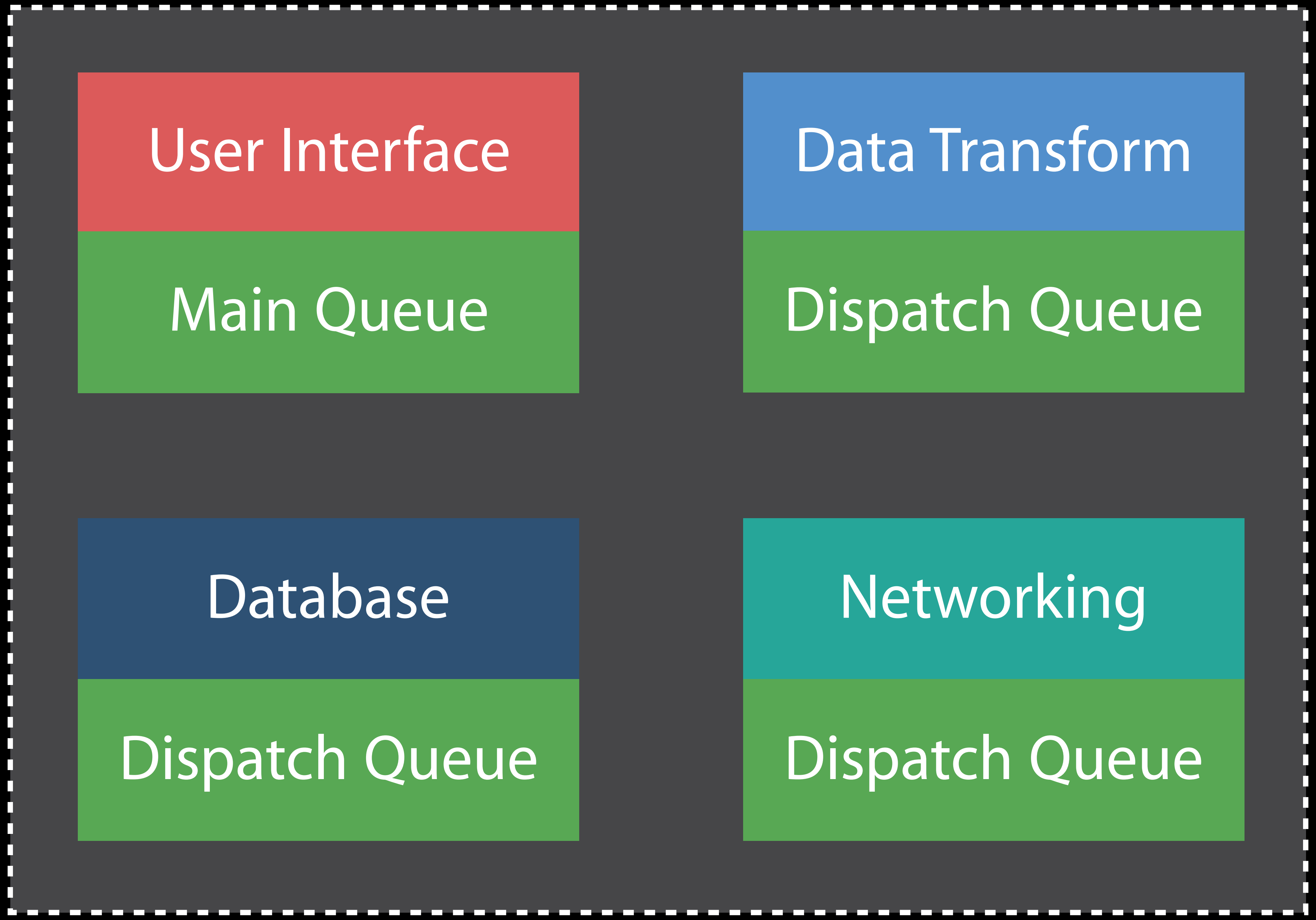


Structuring Your Application

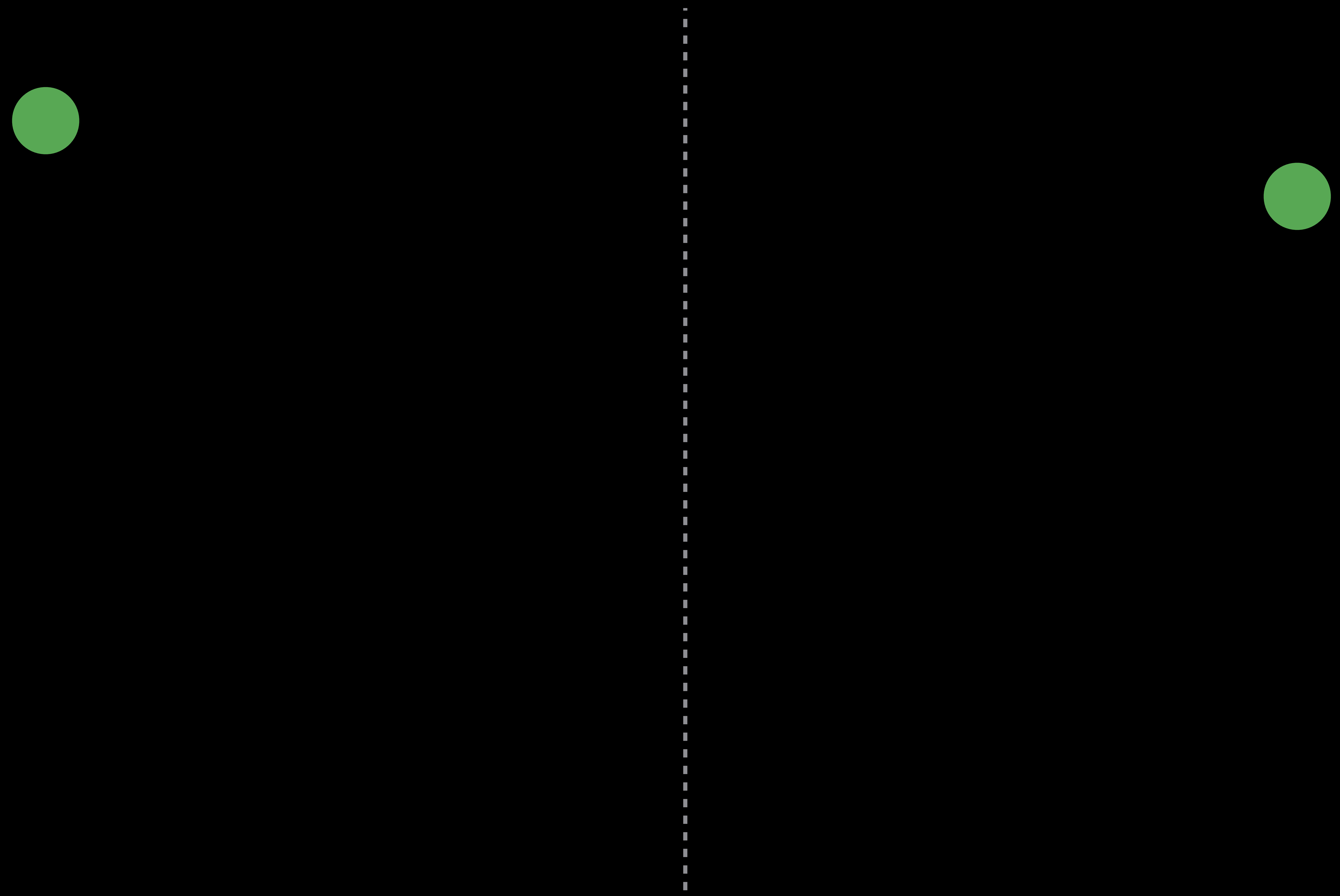
Identify areas of data flow in your application

Split into distinct subsystems

Queues at subsystem granularity



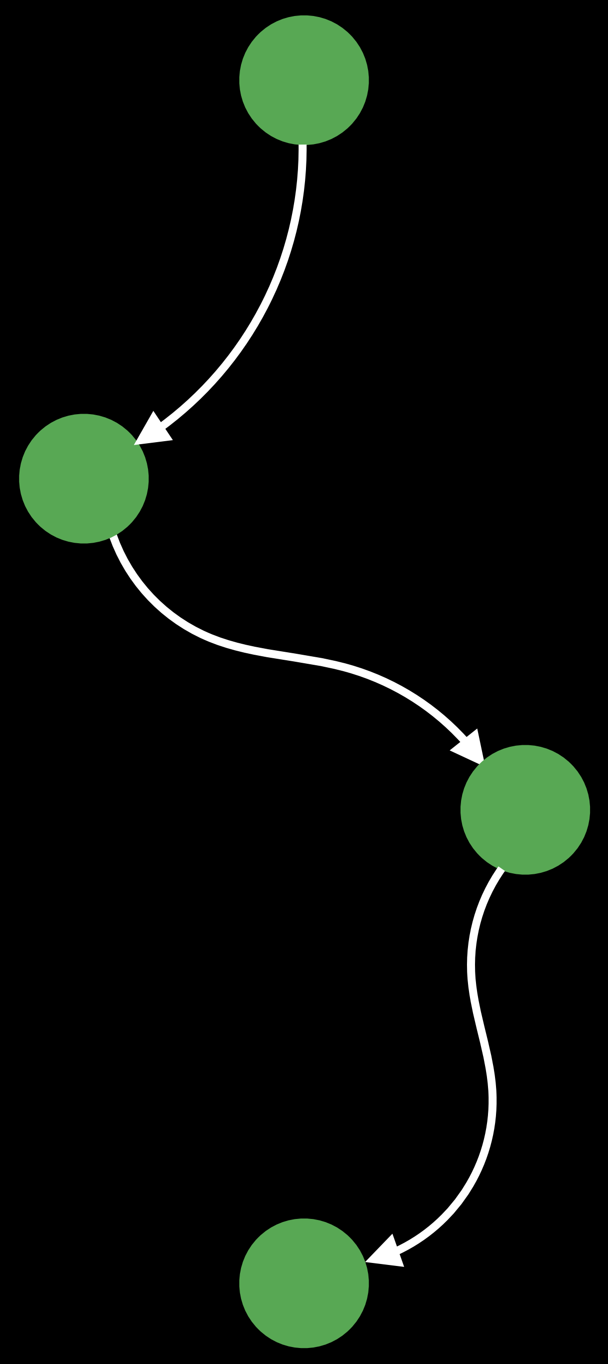
Chaining vs. Grouping Work



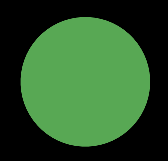
Chaining

Grouping

Chaining vs. Grouping Work

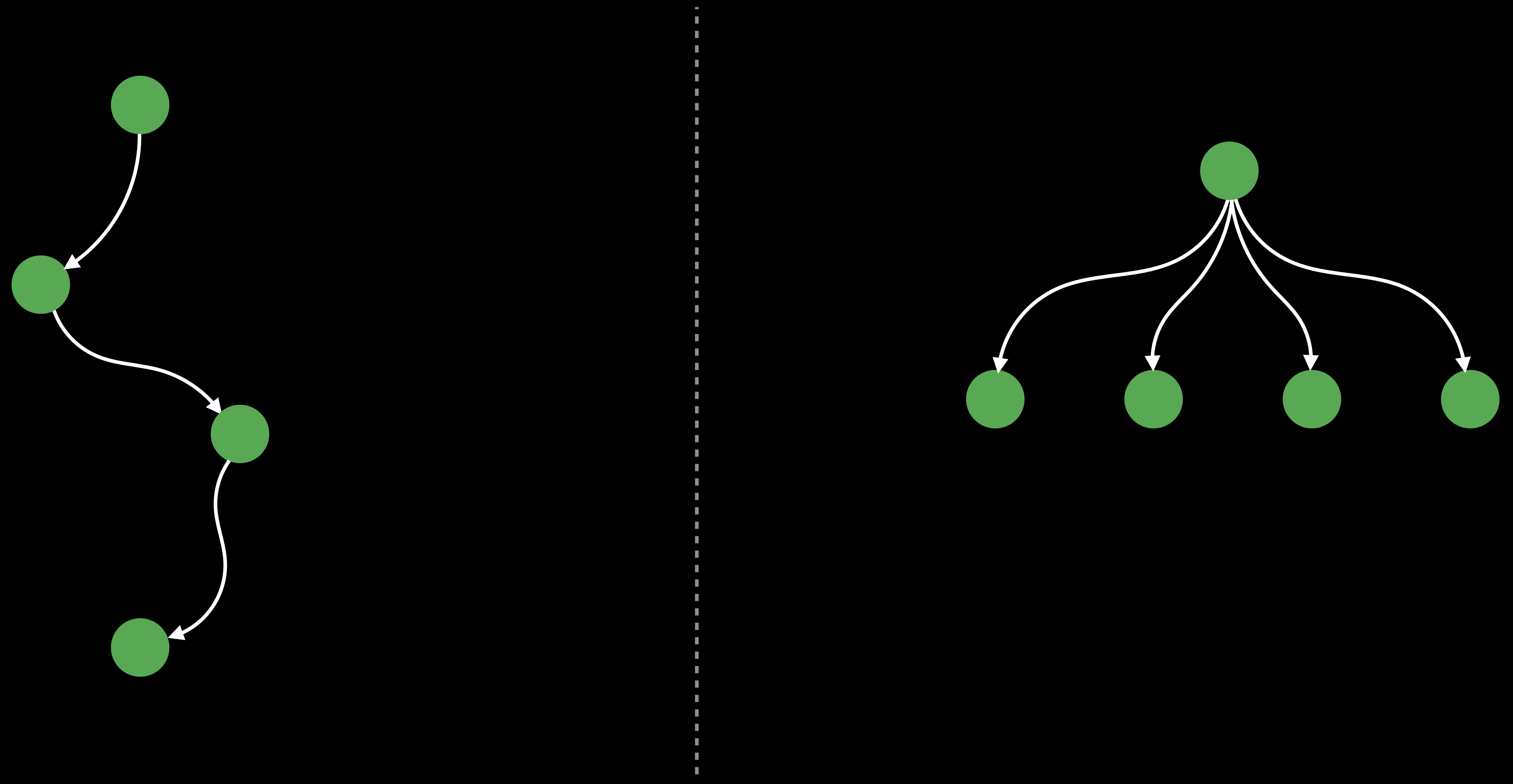


Chaining



Grouping

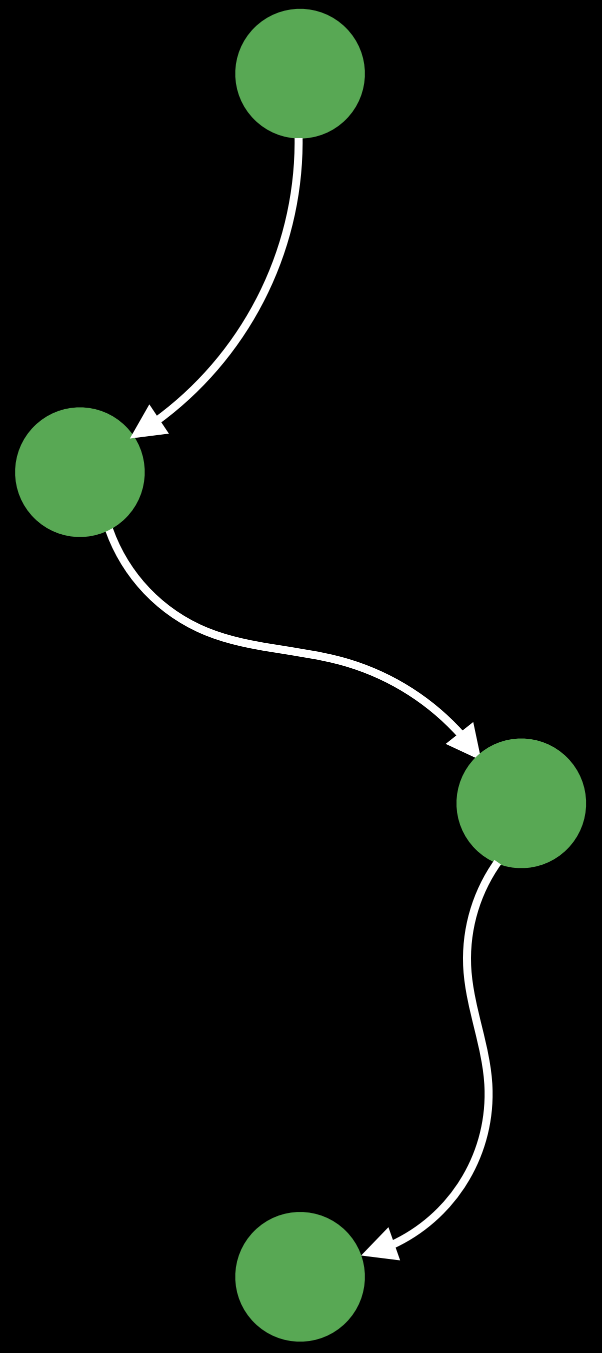
Chaining vs. Grouping Work



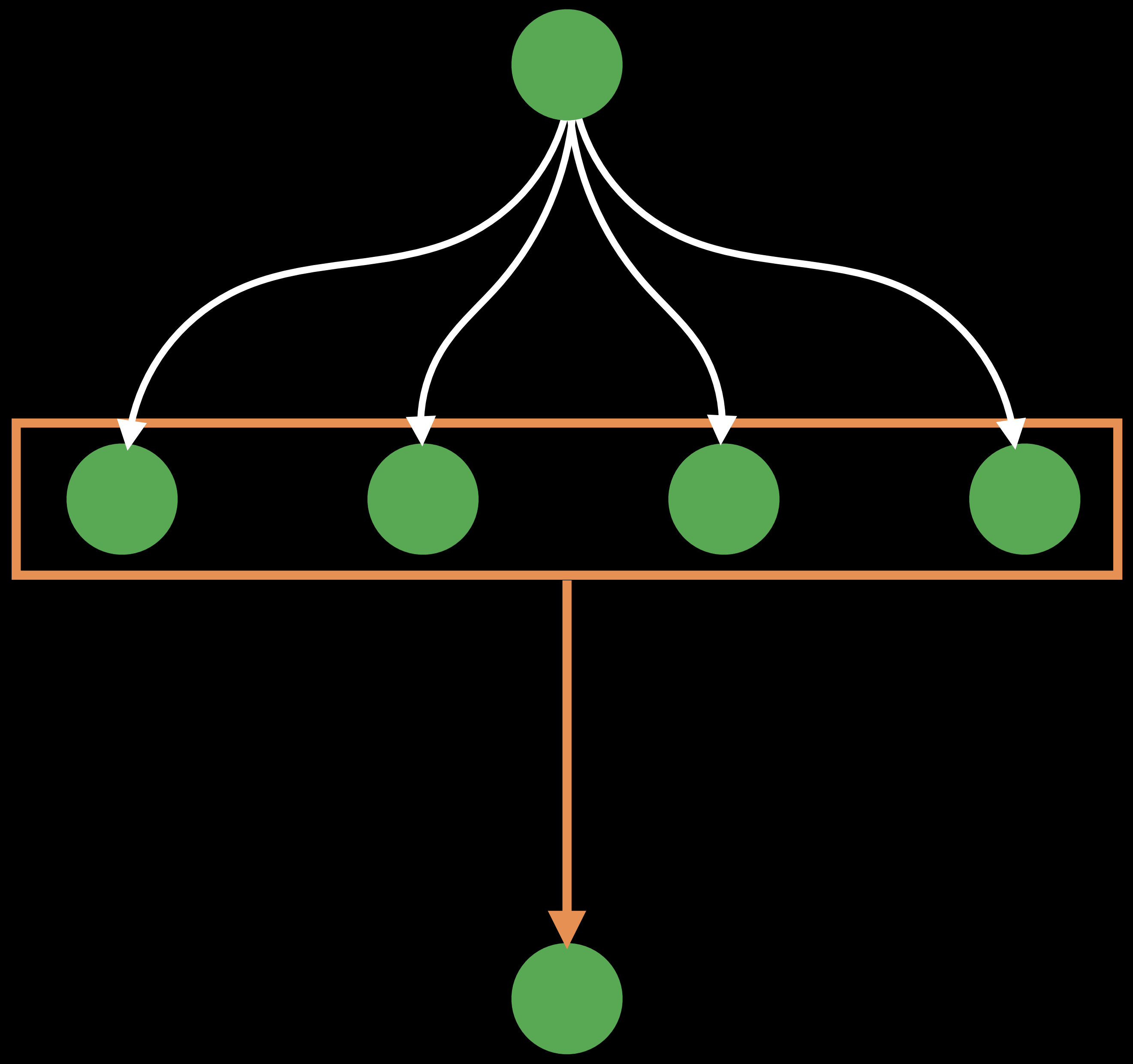
Chaining

Grouping

Chaining vs. Grouping Work

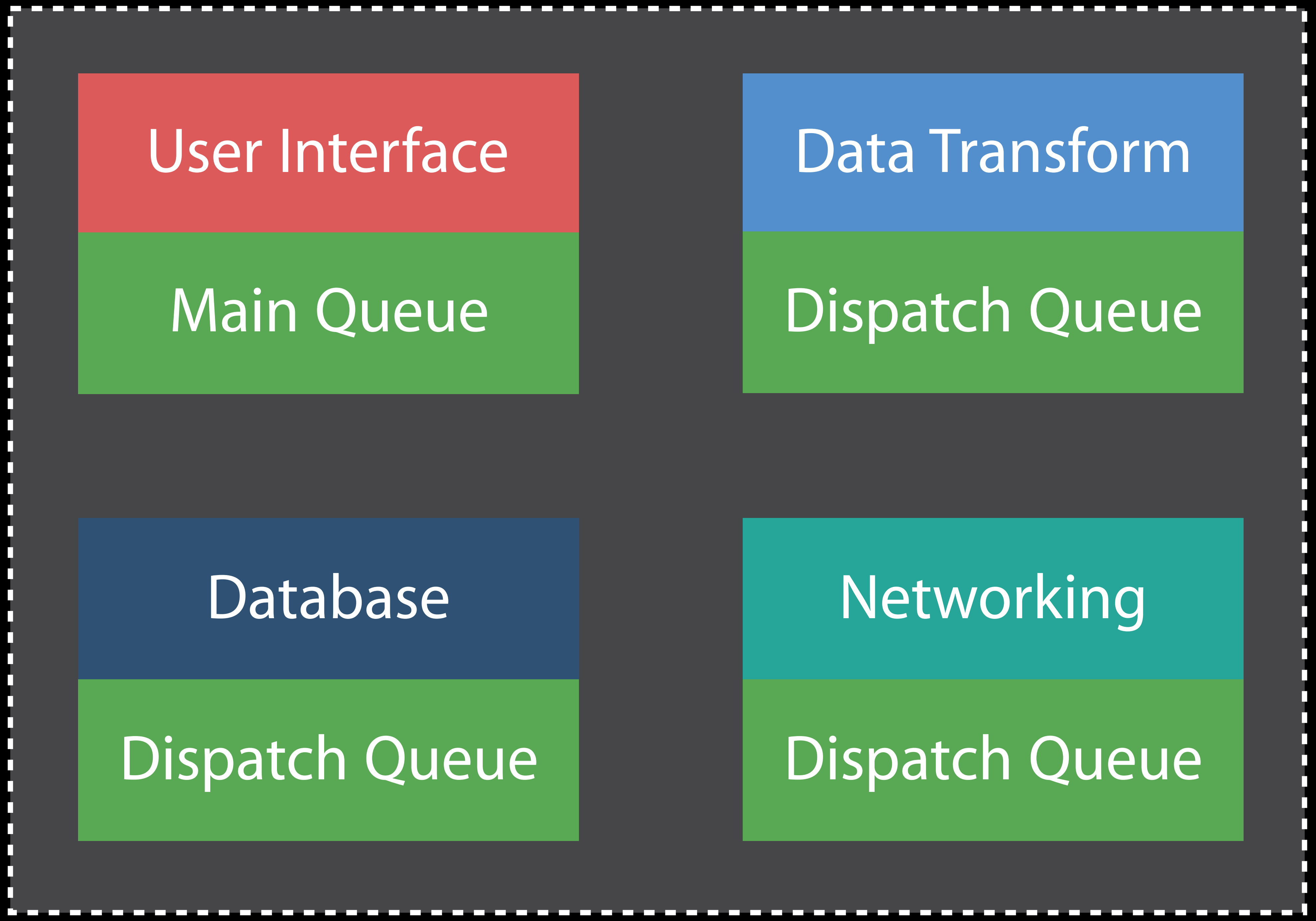


Chaining



Grouping

Grouping Work Together



User Interface

Main Queue

Data Transform

Dispatch Queue

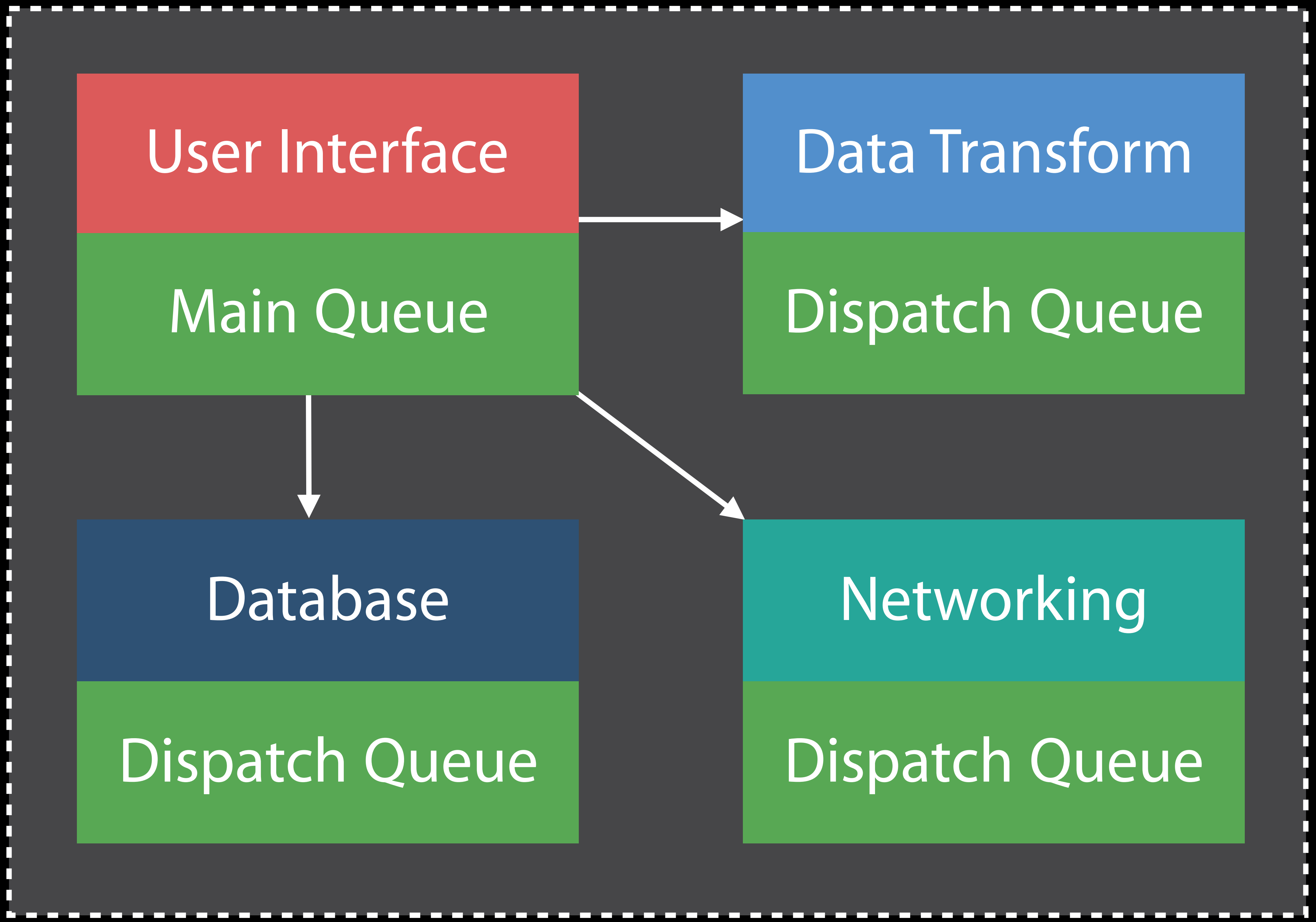
Database

Dispatch Queue

Networking

Dispatch Queue

Grouping Work Together



User Interface

Main Queue

Data Transform

Dispatch Queue

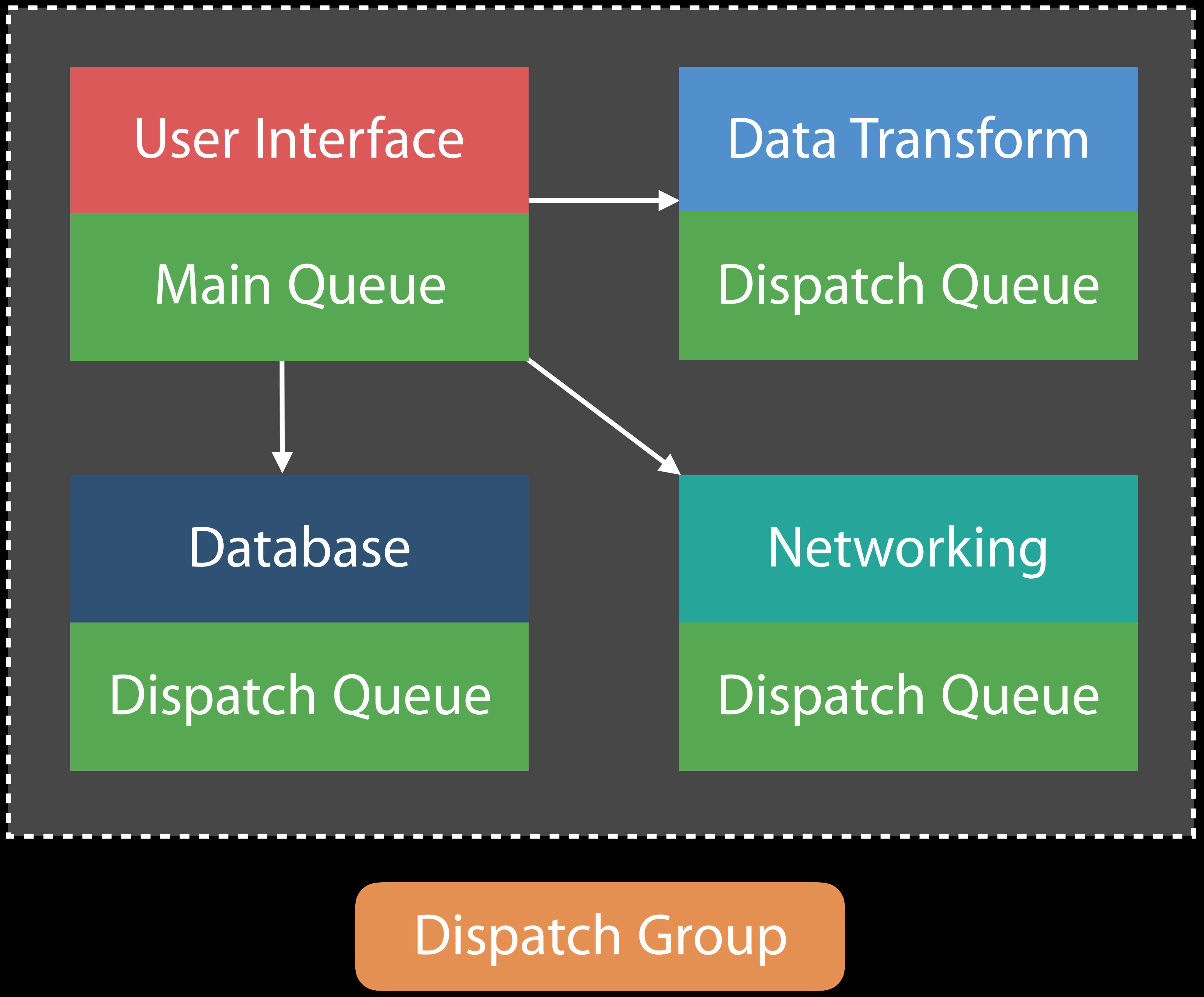
Database

Dispatch Queue

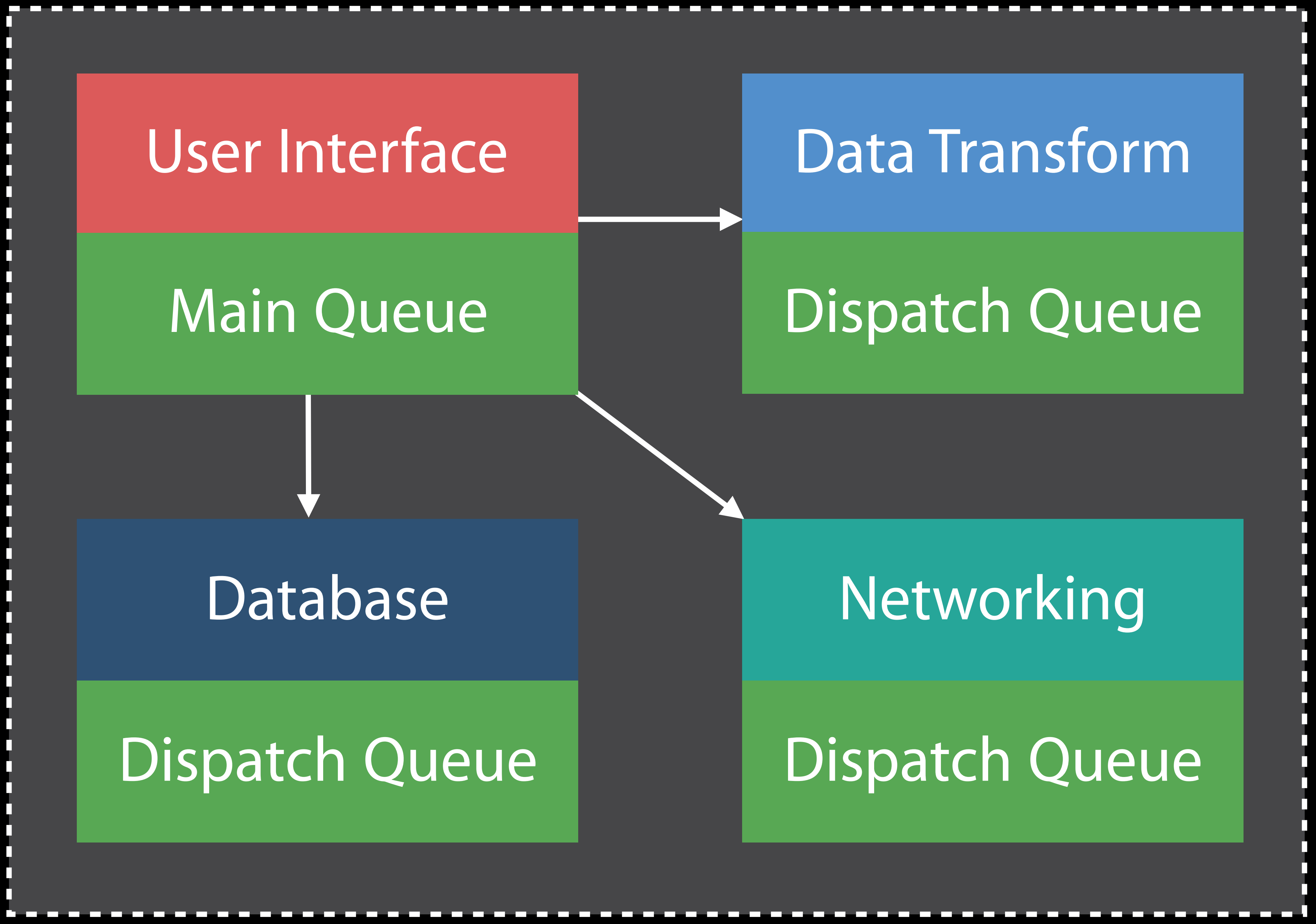
Networking

Dispatch Queue

Grouping Work Together



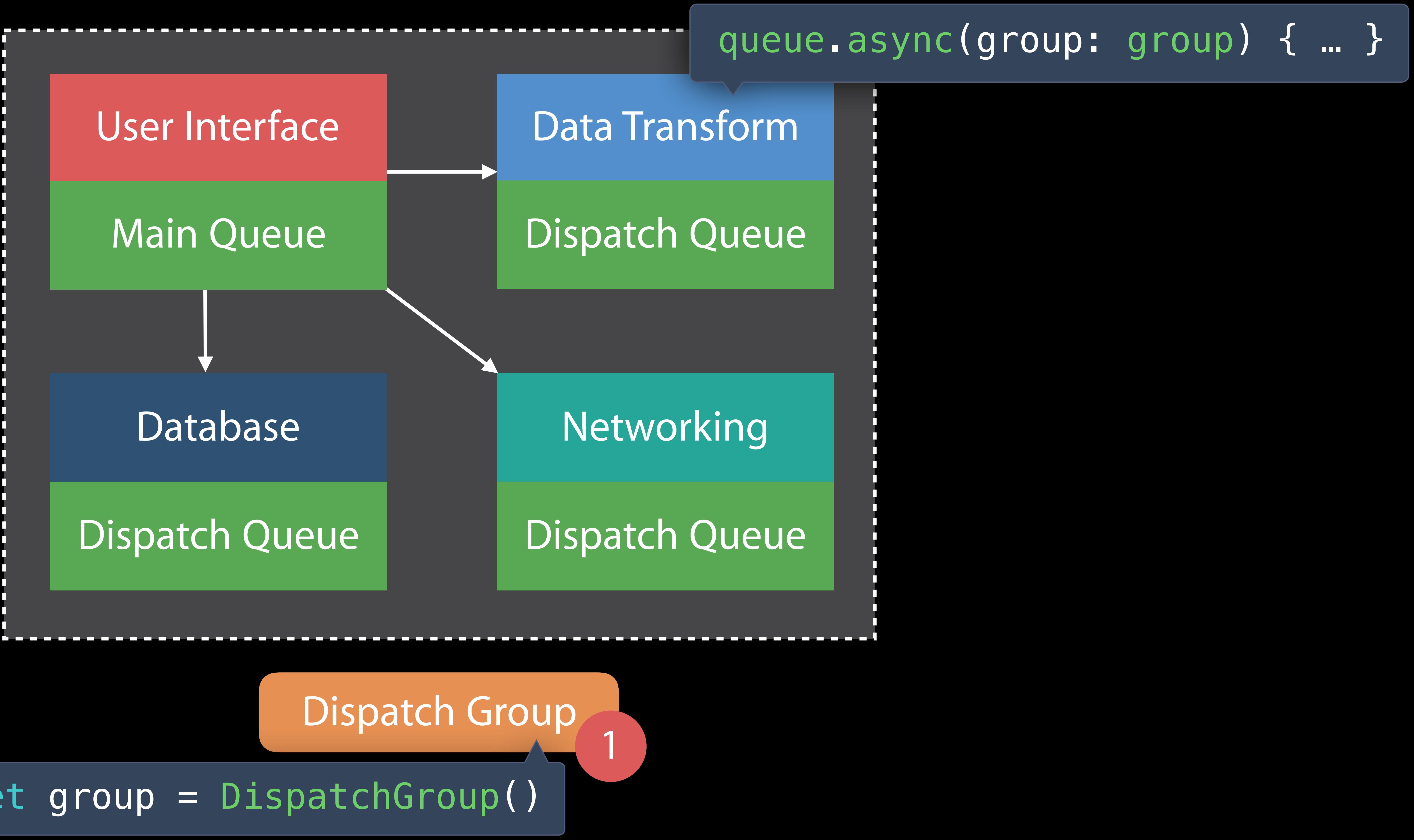
Grouping Work Together



Dispatch Group

```
let group = DispatchGroup()
```

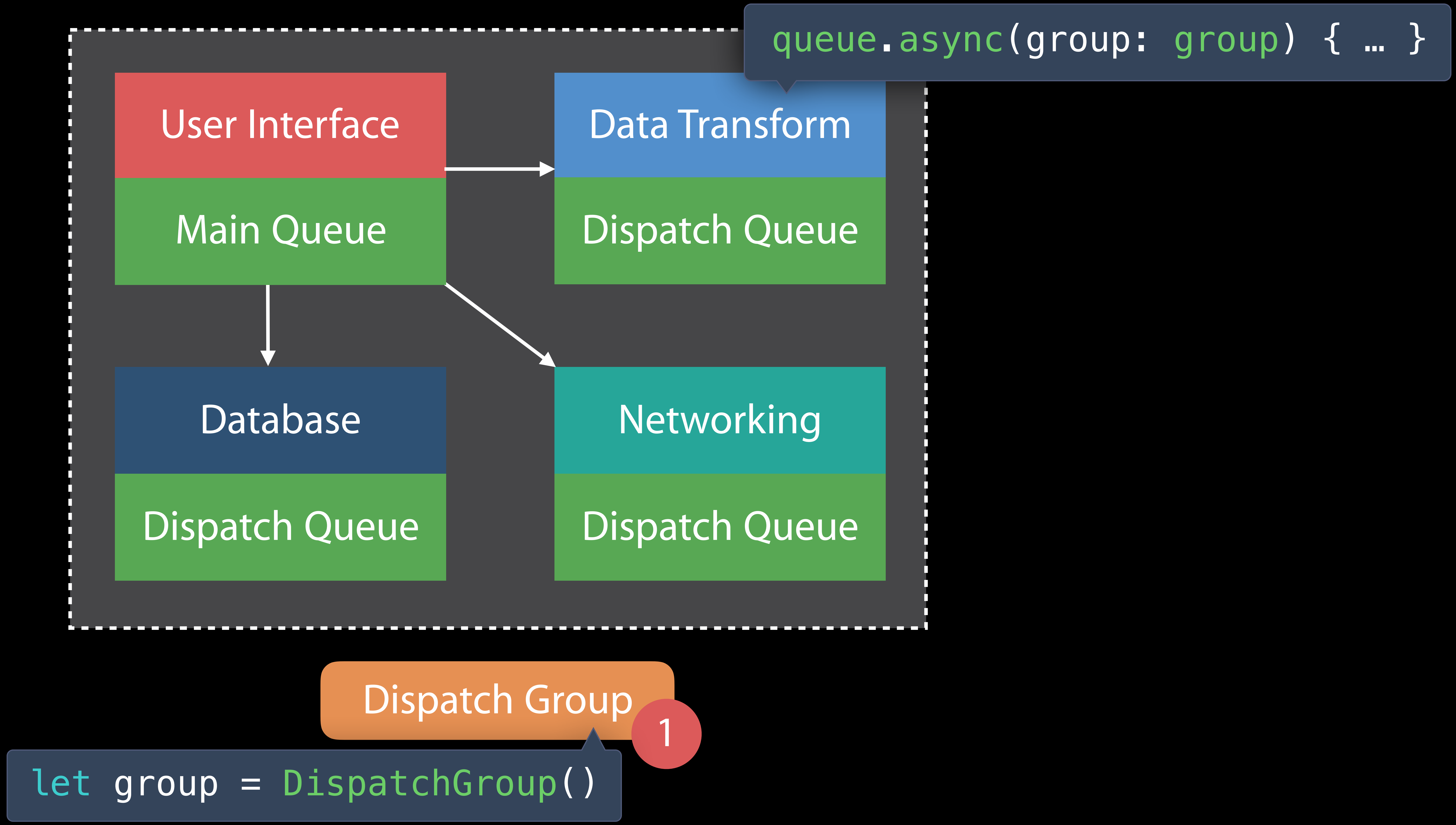
Grouping Work Together



```
queue.async(group: group) { … }
```

```
let group = DispatchGroup()
```

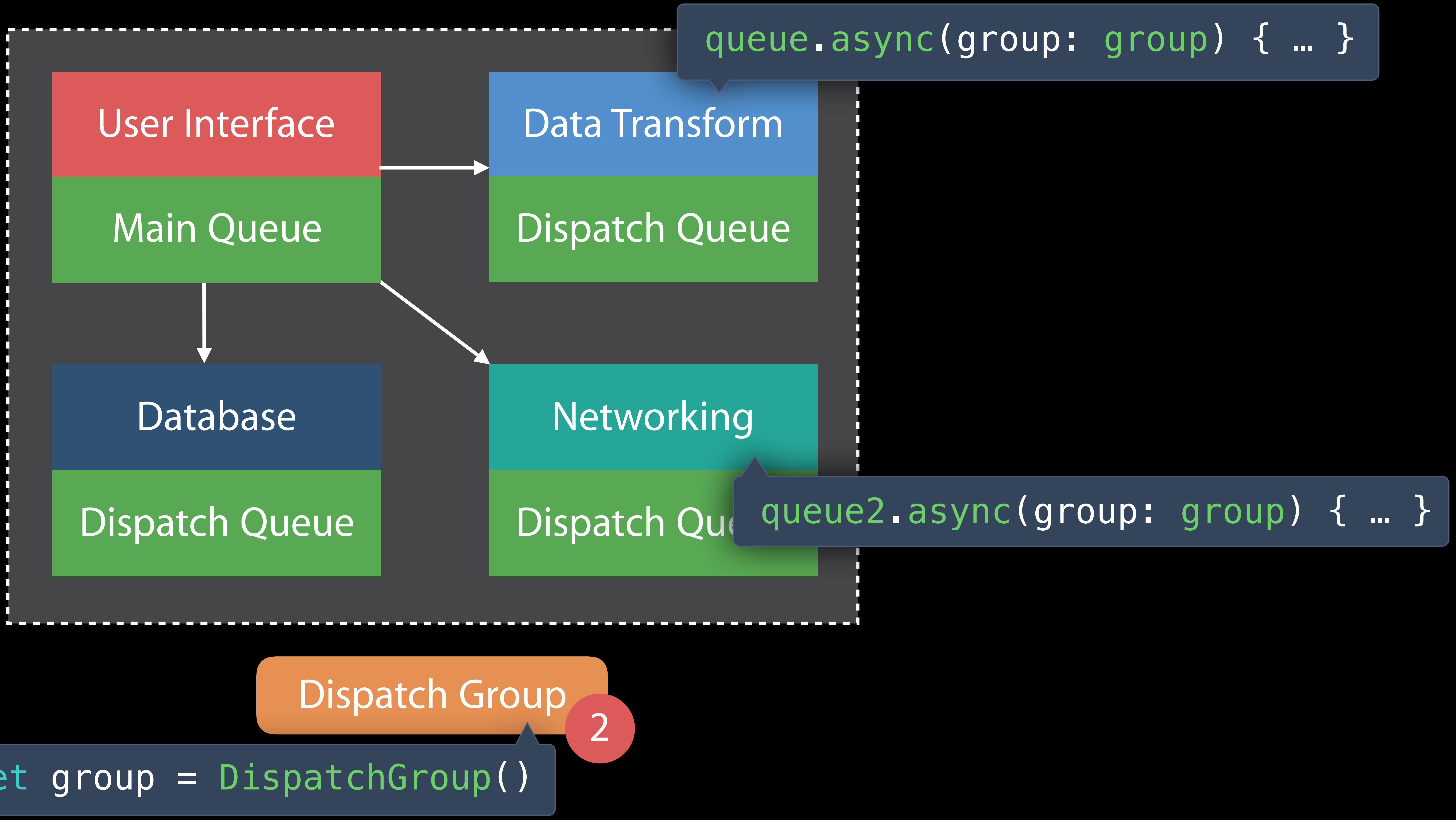
Grouping Work Together



```
queue.async(group: group) { … }
```

```
let group = DispatchGroup()
```

Grouping Work Together

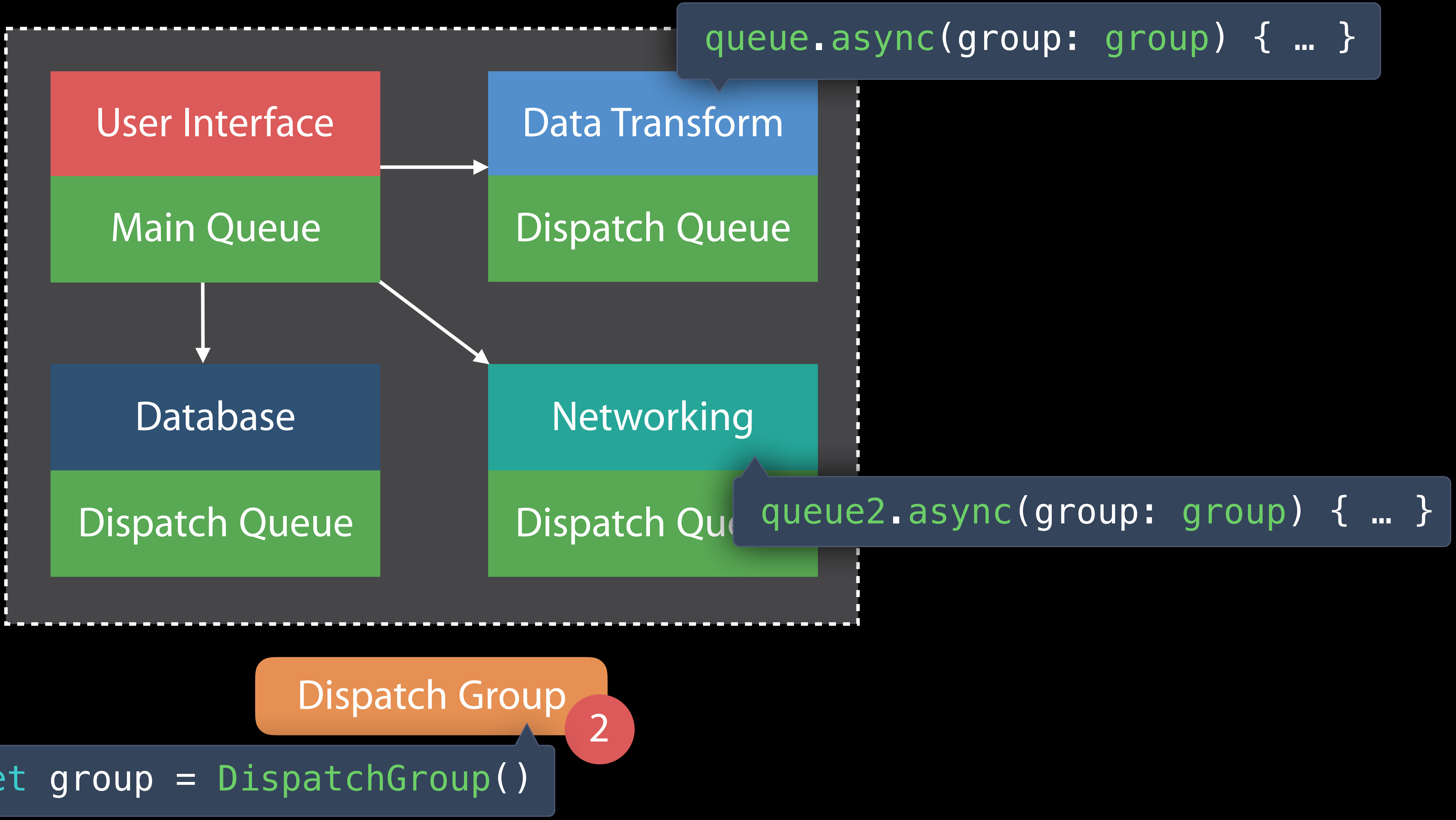


```
queue.async(group: group) { … }
```

```
queue2.async(group: group) { … }
```

```
let group = DispatchGroup()
```

Grouping Work Together

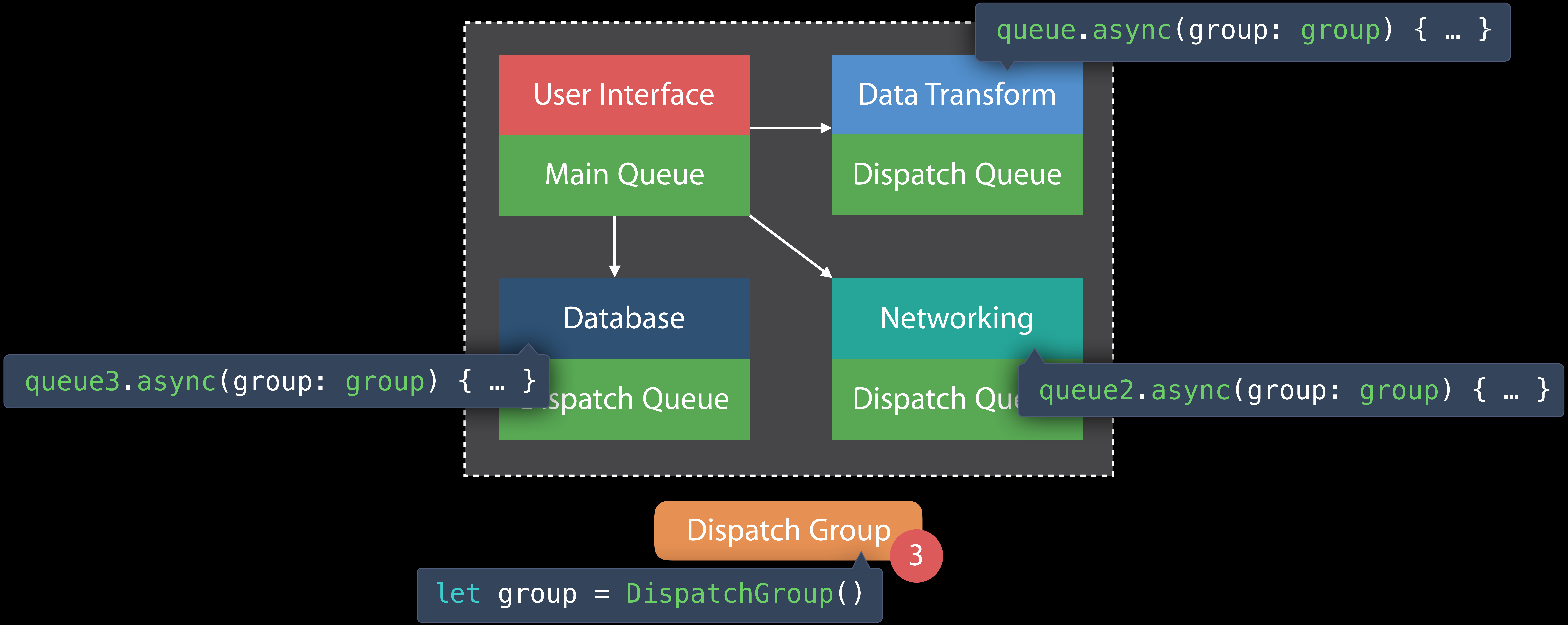


```
queue.async(group: group) { … }
```

```
queue2.async(group: group) { … }
```

```
let group = DispatchGroup()
```

Grouping Work Together



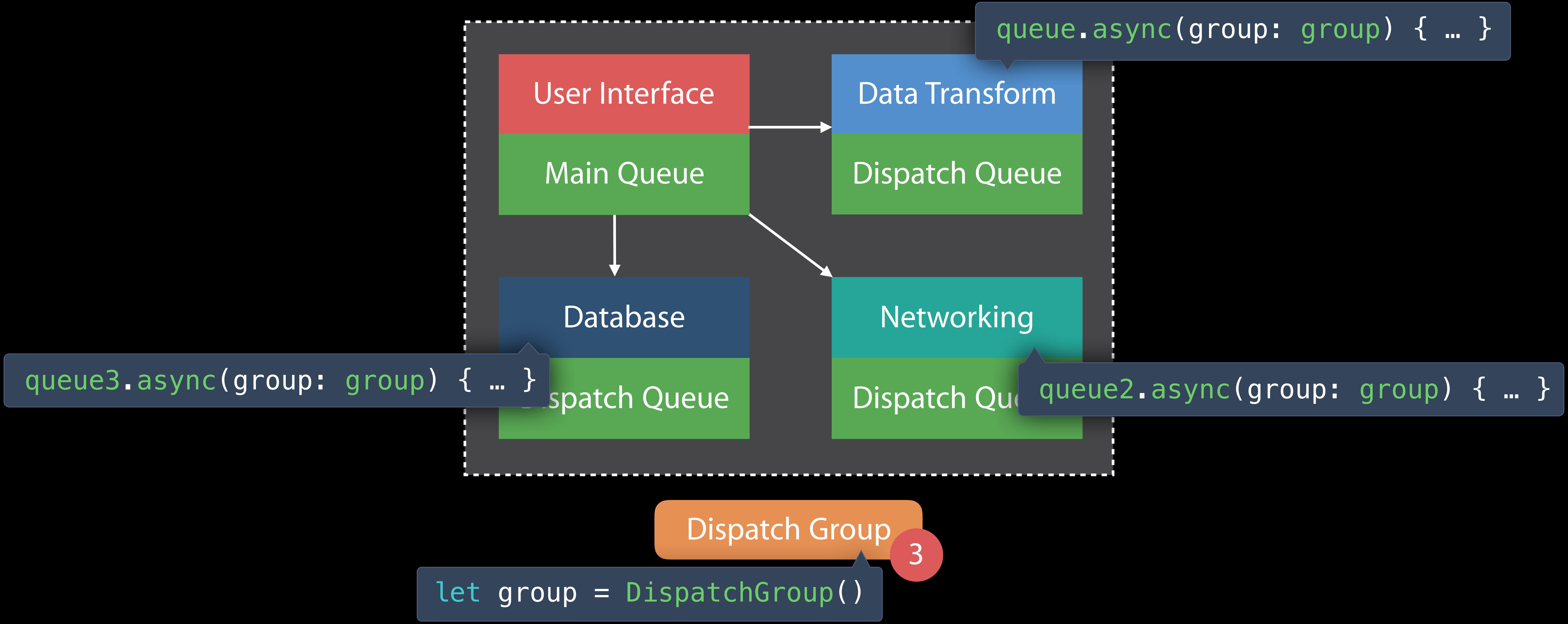
```
queue.async(group: group) { … }
```

```
queue3.async(group: group) { … }
```

```
queue2.async(group: group) { … }
```

```
let group = DispatchGroup()
```

Grouping Work Together



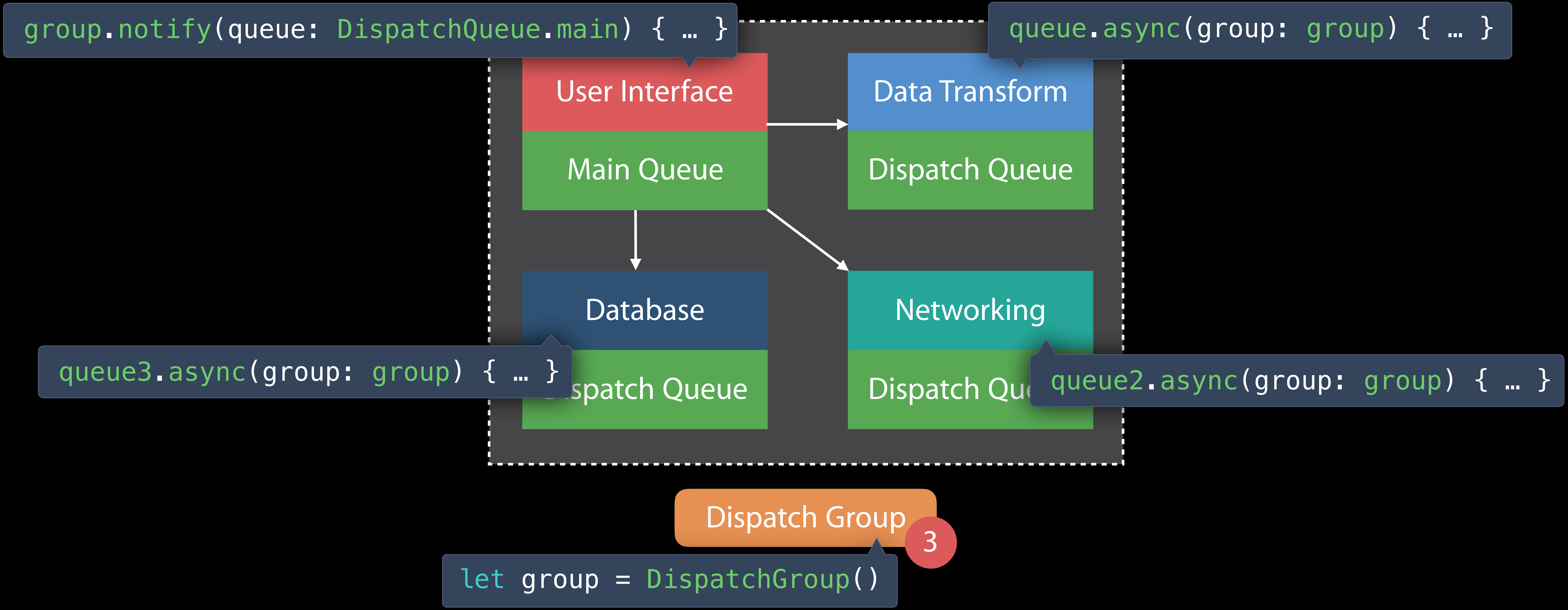
```
queue.async(group: group) { … }
```

```
queue3.async(group: group) { … }
```

```
queue2.async(group: group) { … }
```

```
let group = DispatchGroup()
```

Grouping Work Together



```
group.notify(queue: DispatchQueue.main) { … }
```

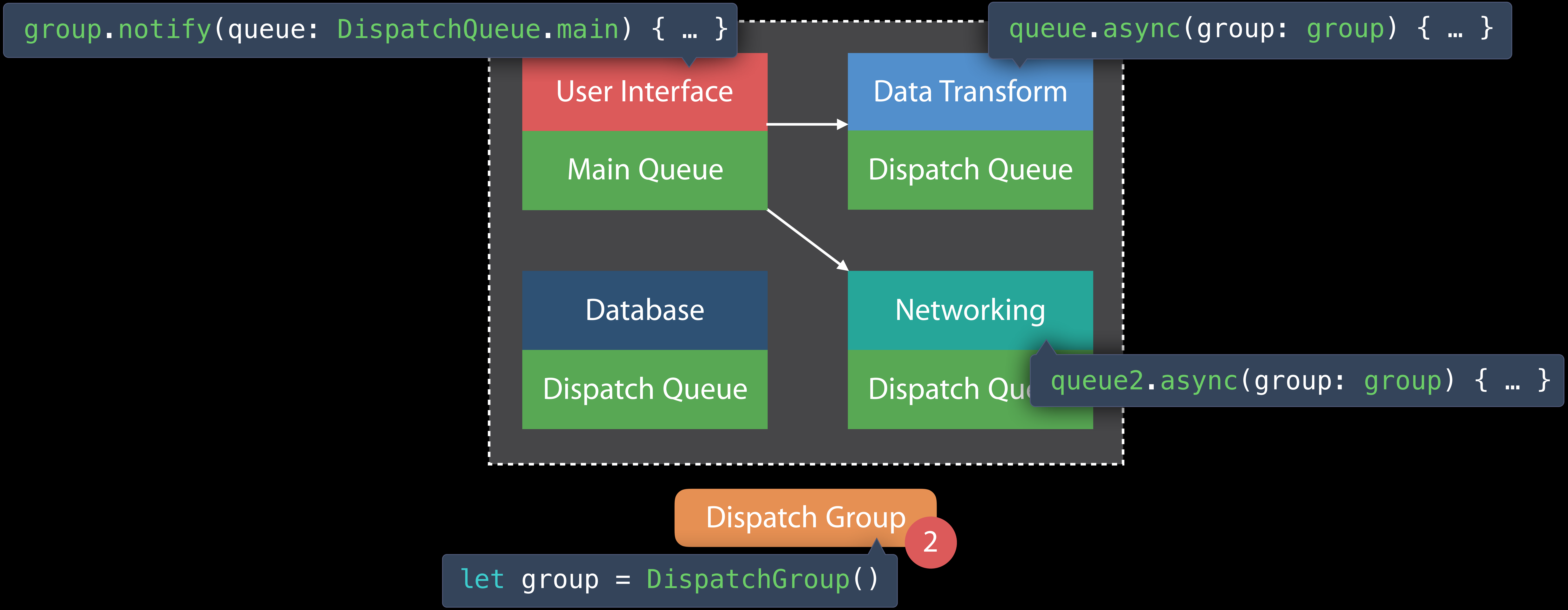
```
queue.async(group: group) { … }
```

```
queue3.async(group: group) { … }
```

```
queue2.async(group: group) { … }
```

```
let group = DispatchGroup()
```

Grouping Work Together



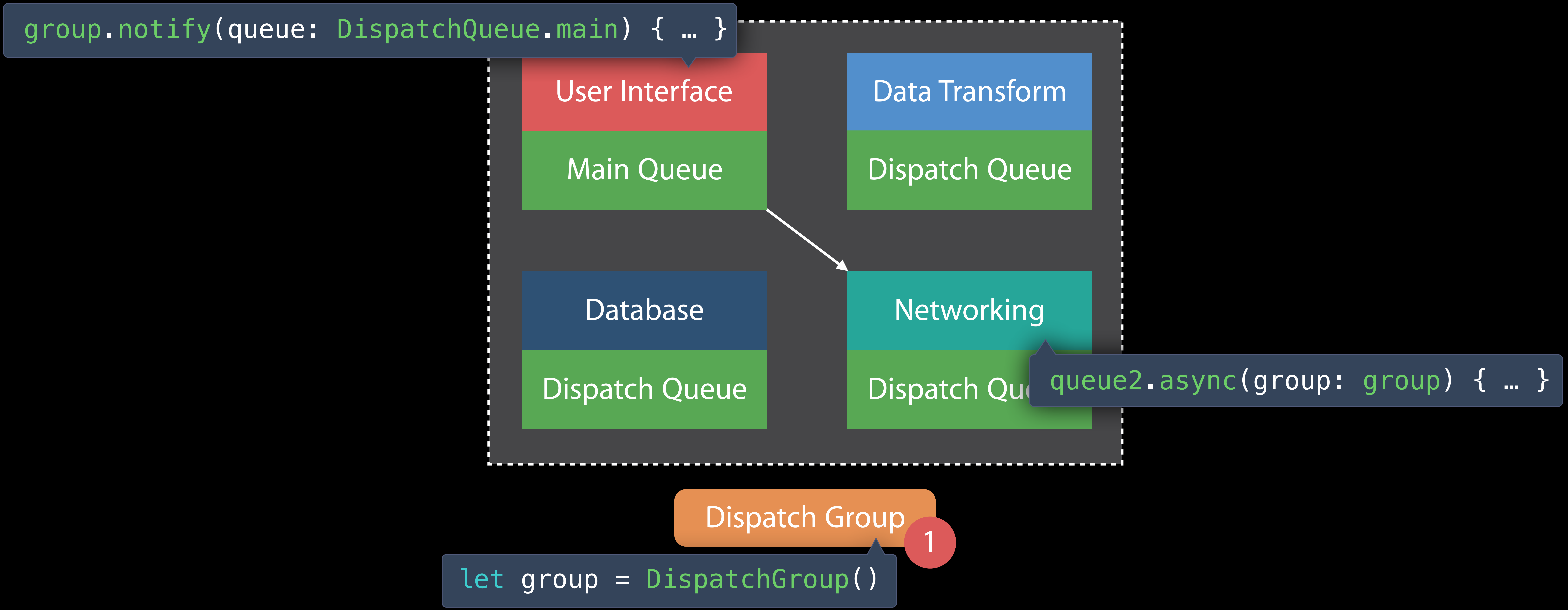
```
group.notify(queue: DispatchQueue.main) { … }
```

```
queue.async(group: group) { … }
```

```
queue2.async(group: group) { … }
```

```
let group = DispatchGroup()
```

Grouping Work Together



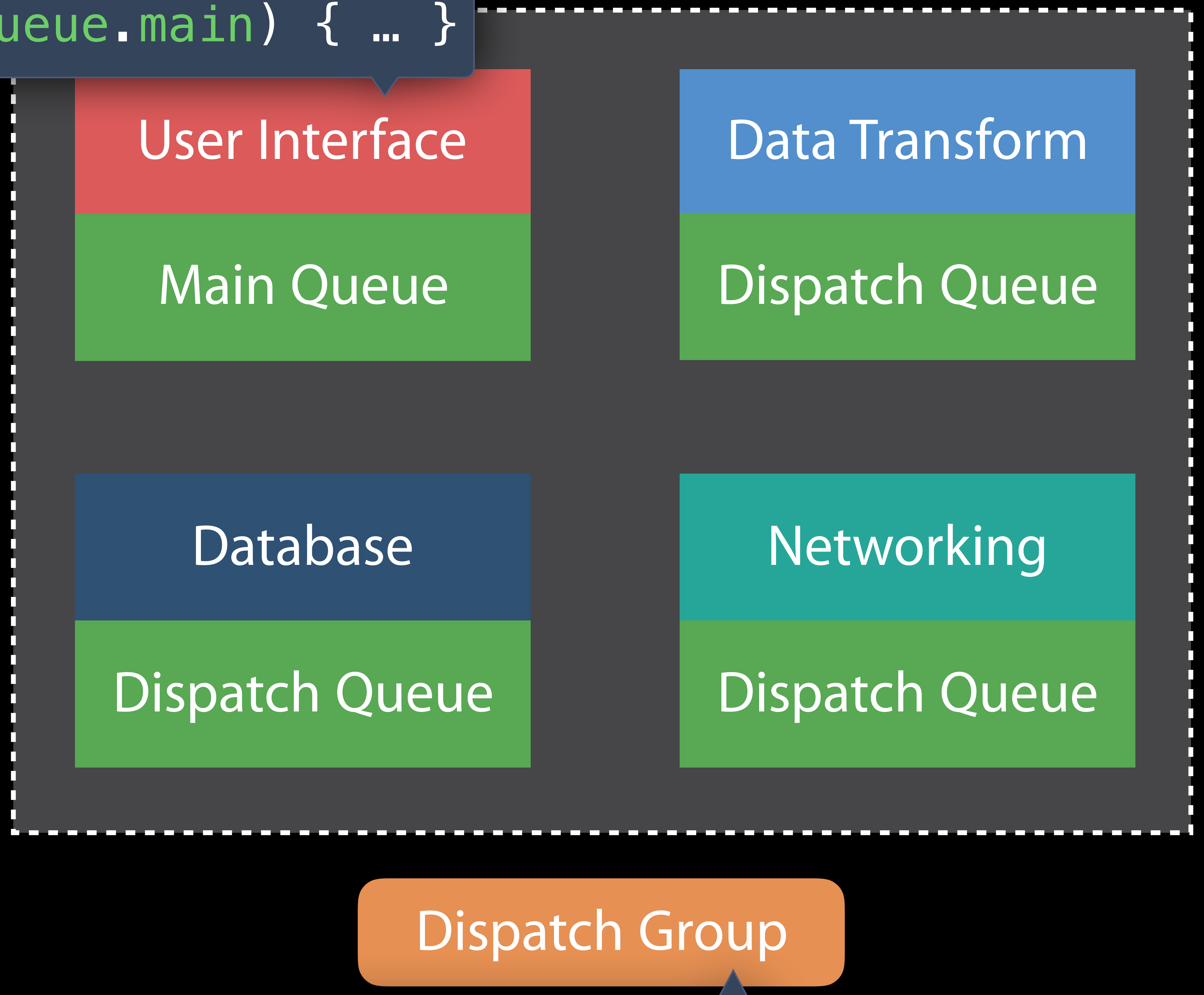
```
group.notify(queue: DispatchQueue.main) { … }
```

```
queue2.async(group: group) { … }
```

```
let group = DispatchGroup()
```

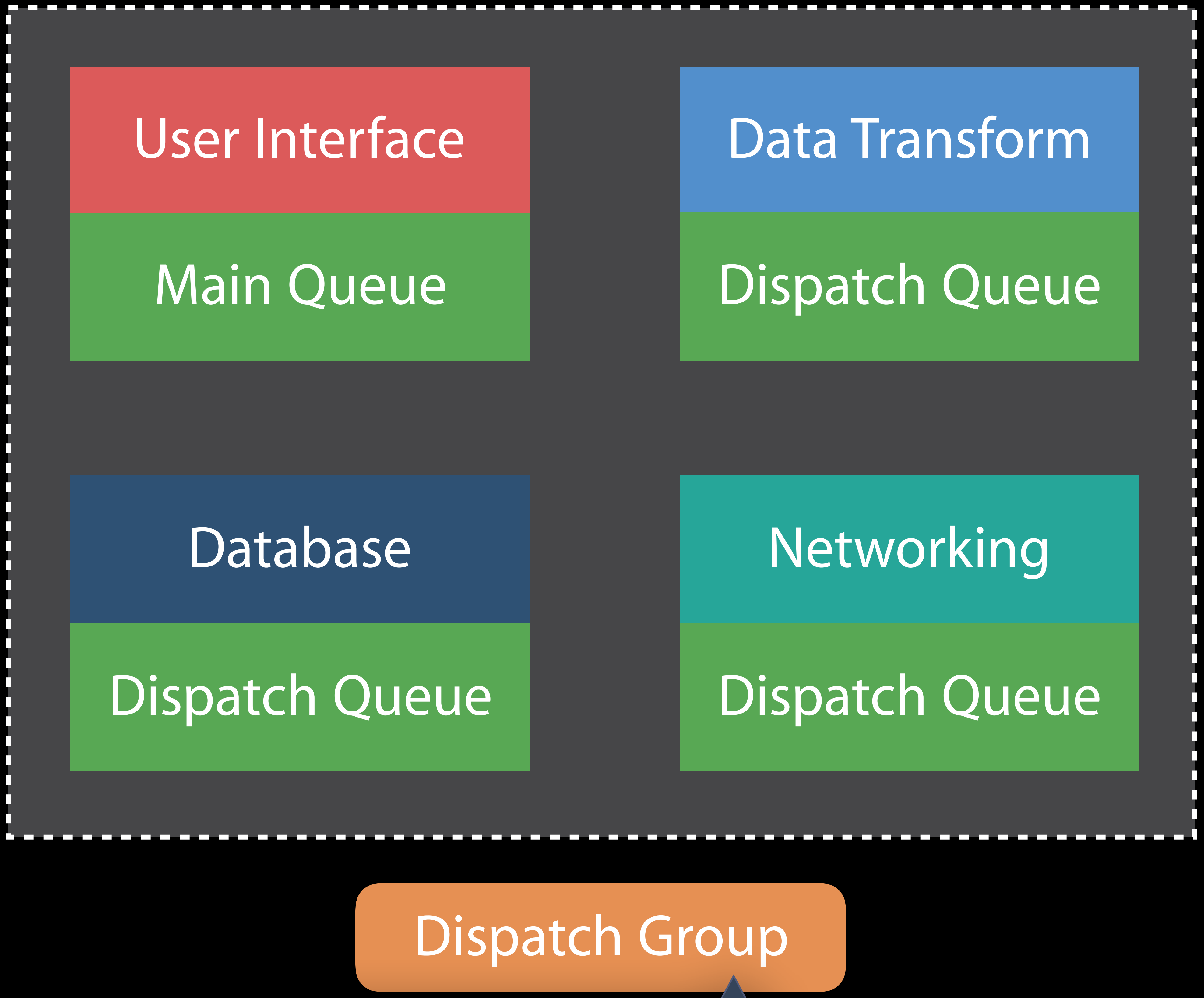
Grouping Work Together

```
group.notify(queue: DispatchQueue.main) { … }
```



```
let group = DispatchGroup()
```

Grouping Work Together



```
let group = DispatchGroup()
```

Synchronizing Between Subsystems

Can use subsystem serial queues for mutual exclusion

Synchronizing Between Subsystems

Can use subsystem serial queues for mutual exclusion

Use `.sync` to safely access properties from subsystems

```
var count: Int {
    queue.sync { self.connections.count }
}
```

Synchronizing Between Subsystems

Can use subsystem serial queues for mutual exclusion

Use `.sync` to safely access properties from subsystems

Be aware of "lock ordering" introduced between subsystems

```
var count: Int {
    queue.sync { self.connections.count }
}
```

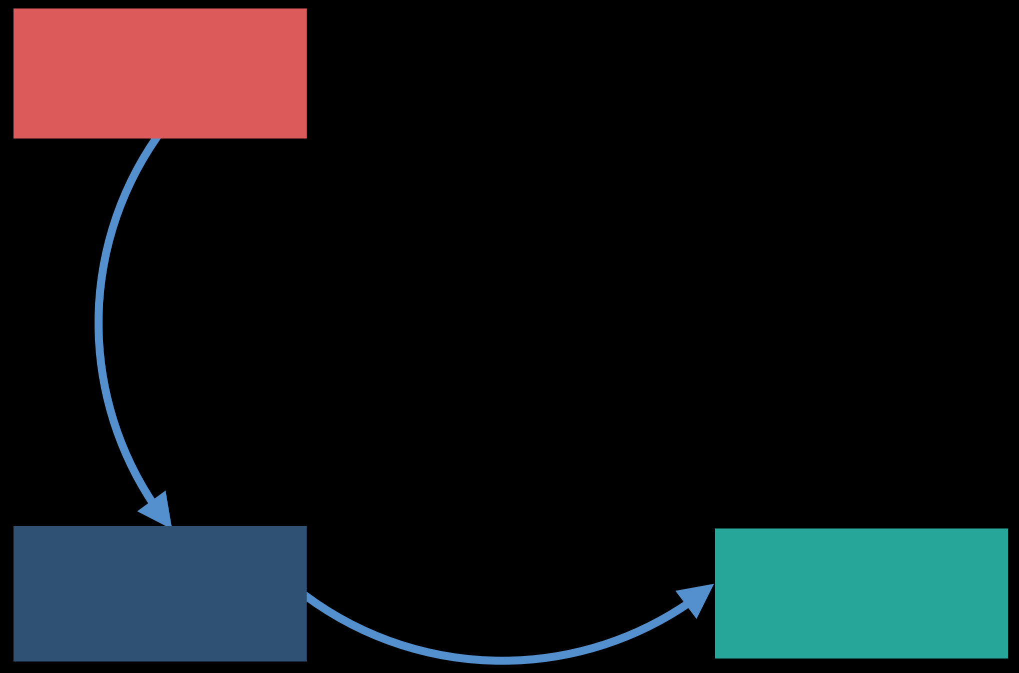
Synchronizing Between Subsystems

Can use subsystem serial queues for mutual exclusion

Use `.sync` to safely access properties from subsystems

Be aware of “lock ordering” introduced between subsystems

```
var count: Int {
    queue.sync { self.connections.count }
}
```



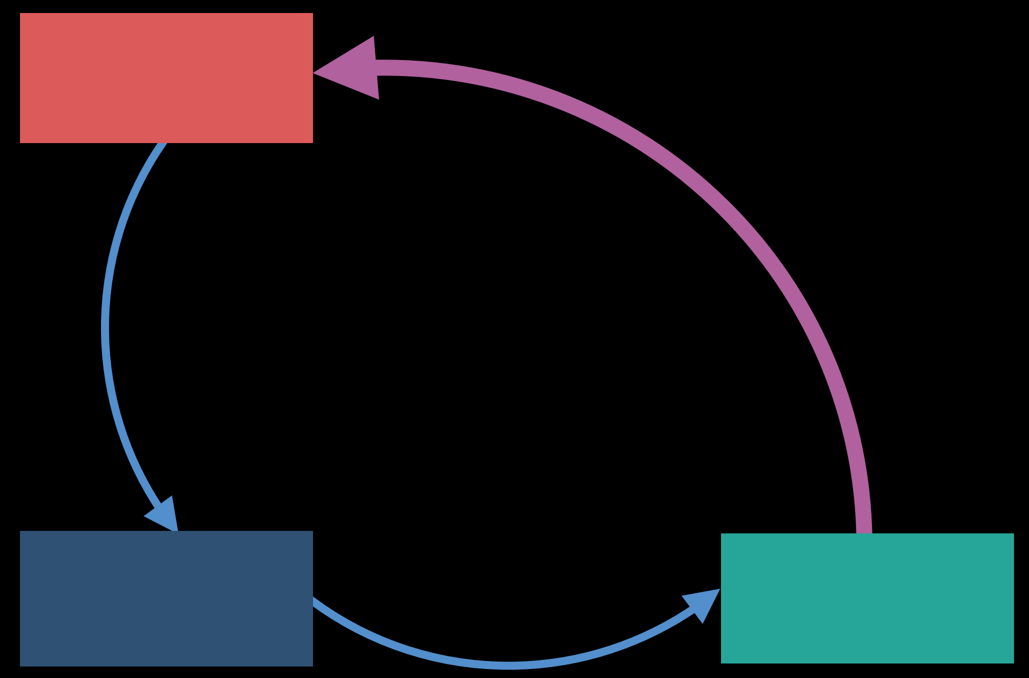
Synchronizing Between Subsystems

Can use subsystem serial queues for mutual exclusion

Use `.sync` to safely access properties from subsystems

Be aware of “lock ordering” introduced between subsystems

```
var count: Int {
    queue.sync { self.connections.count }
}
```



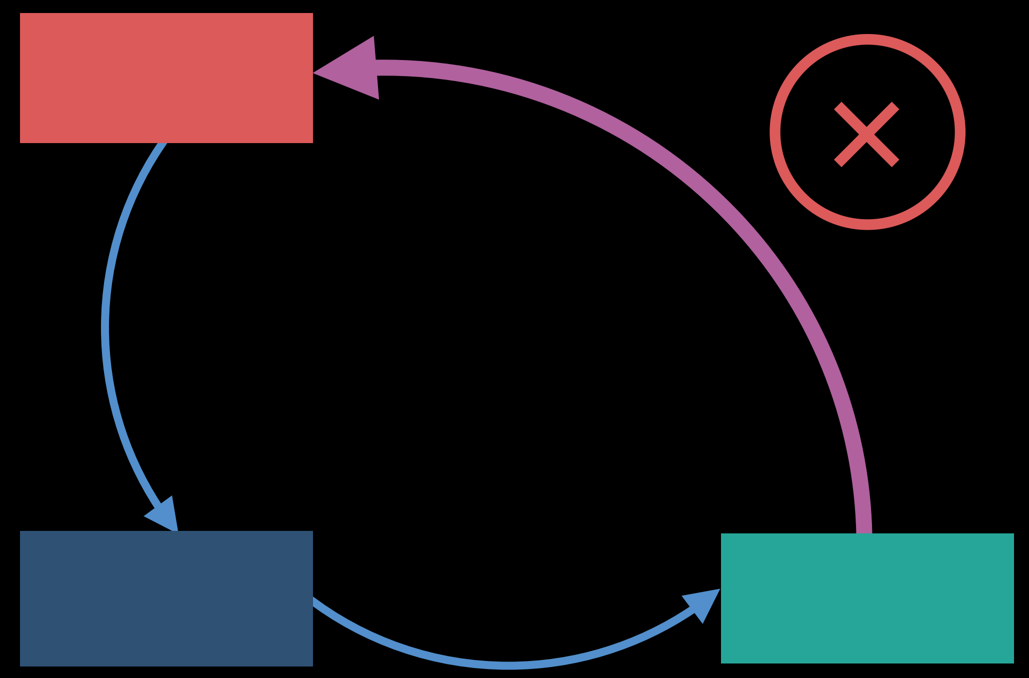
Synchronizing Between Subsystems

Can use subsystem serial queues for mutual exclusion

Use `.sync` to safely access properties from subsystems

Be aware of “lock ordering” introduced between subsystems

```
var count: Int {
    queue.sync { self.connections.count }
}
```



Dispatch Inside Subsystems

Choosing a Quality of Service

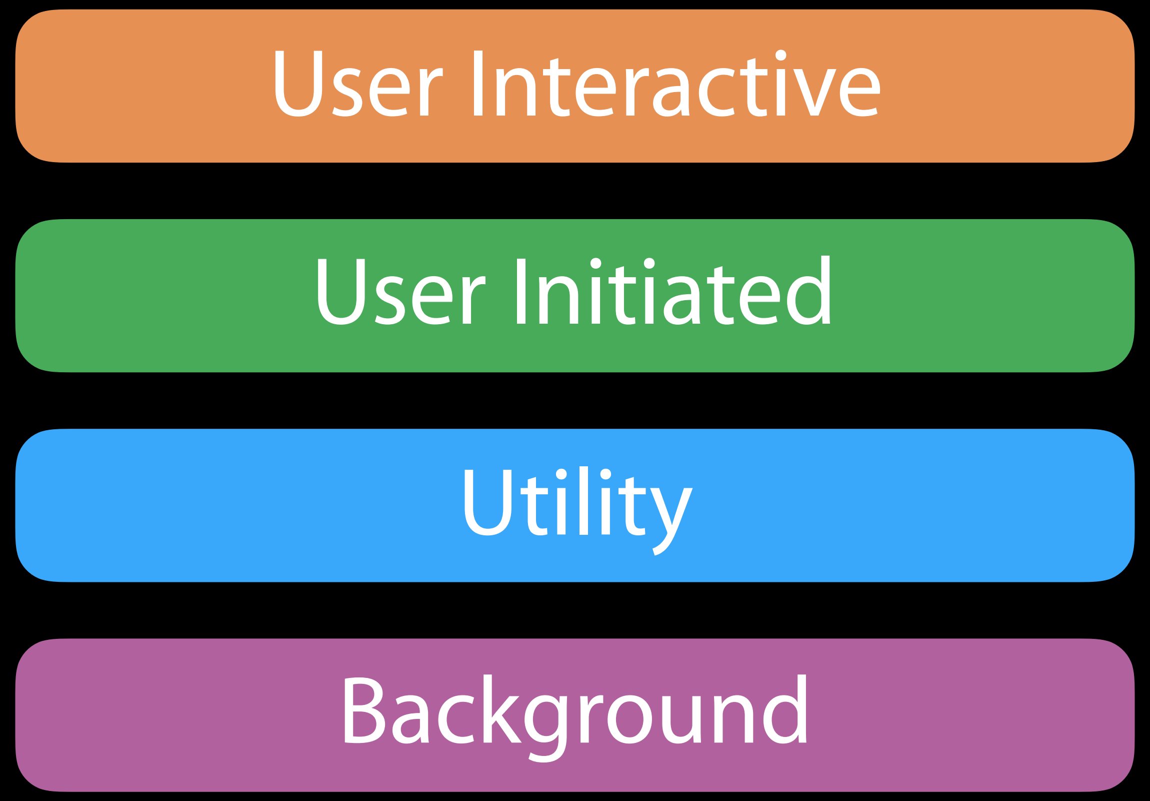
QoS provides explicit classification of work

- User Interactive
- User Initiated
- Utility
- Background

Choosing a Quality of Service

QoS provides explicit classification of work

Indicates developer intent



User Interactive

User Initiated

Utility

Background

Choosing a Quality of Service

QoS provides explicit classification of work

Indicates developer intent

Affects execution properties of your work

User Interactive

User Initiated

Utility

Background

Choosing a Quality of Service

QoS provides explicit classification of work

Indicates developer intent

Affects execution properties of your work

- User Interactive
- User Initiated
- Utility
- Background

Using Quality of Service Classes

```
queue.async(qos: .background) {
    print("Maintenance work")
}

queue.async(qos: .userInitiated) {
    print(“Button tapped”)
}
```

Using Quality of Service Classes

Use `.async` to submit work with a specific QoS class

```
queue.async(qos: .background) {
    print("Maintenance work")
}

queue.async(qos: .userInitiated) {
    print(“Button tapped”)
}
```

Using Quality of Service Classes

Use `.async` to submit work with a specific QoS class

Dispatch helps resolve priority inversions

```swift
queue.async(qos: .background) {
    print("Maintenance work")
}

queue.async(qos: .userInitiated) {
    print(“Button tapped”)
}
```

Using Quality of Service Classes

Use `.async` to submit work with a specific QoS class

Dispatch helps resolve priority inversions

Create single-purpose queues with a specific QoS class

```
queue.async(qos: .background) {
    print("Maintenance work")
}

queue.async(qos: .userInitiated) {
    print(“Button tapped”)
}
```

DispatchWorkItem

By default `.async` captures execution context at time of submission

DispatchWorkItem

By default `.async` captures execution context at time of submission

Create `DispatchWorkItem` from closures to control execution properties

```
let item = DispatchWorkItem(flags: .assignCurrentContext) {
    print("Hello WWDC 2016!")
}

queue.async(execute: item)
```

DispatchWorkItem

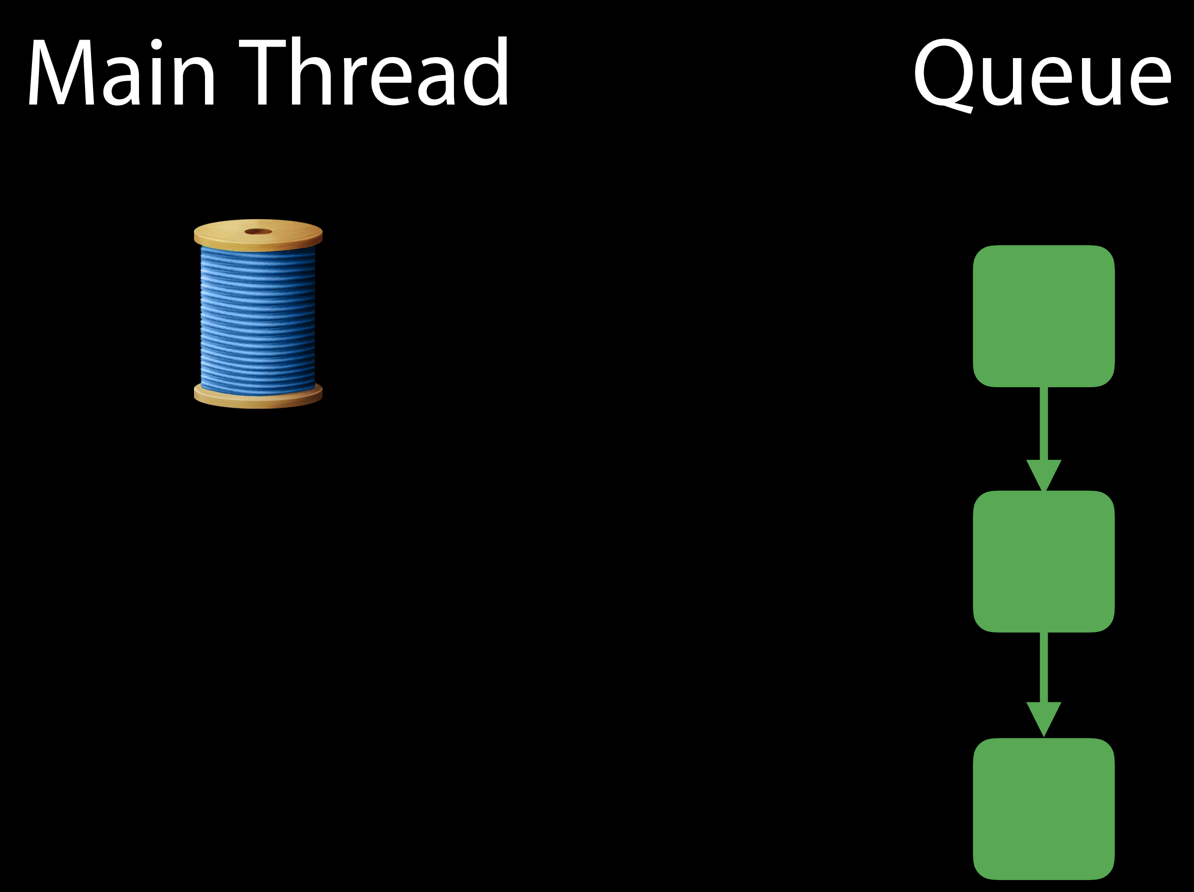
By default `.async` captures execution context at time of submission

Create `DispatchWorkItem` from closures to control execution properties

Use `.assignCurrentContext` to capture current QoS at time of creation

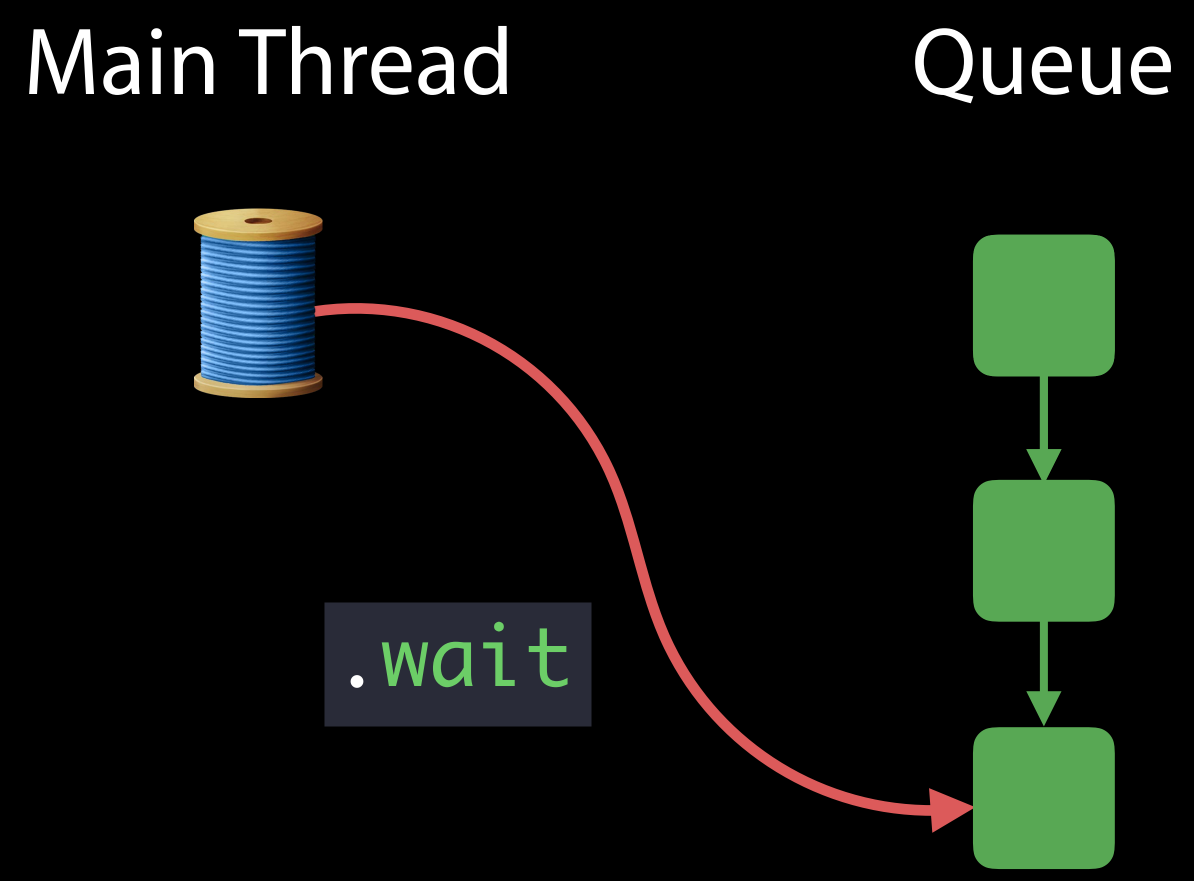
```
let item = DispatchWorkItem(flags: .assignCurrentContext) {
    print("Hello WWDC 2016!")
}

queue.async(execute: item)
```

Waiting for Work Items



Waiting for Work Items

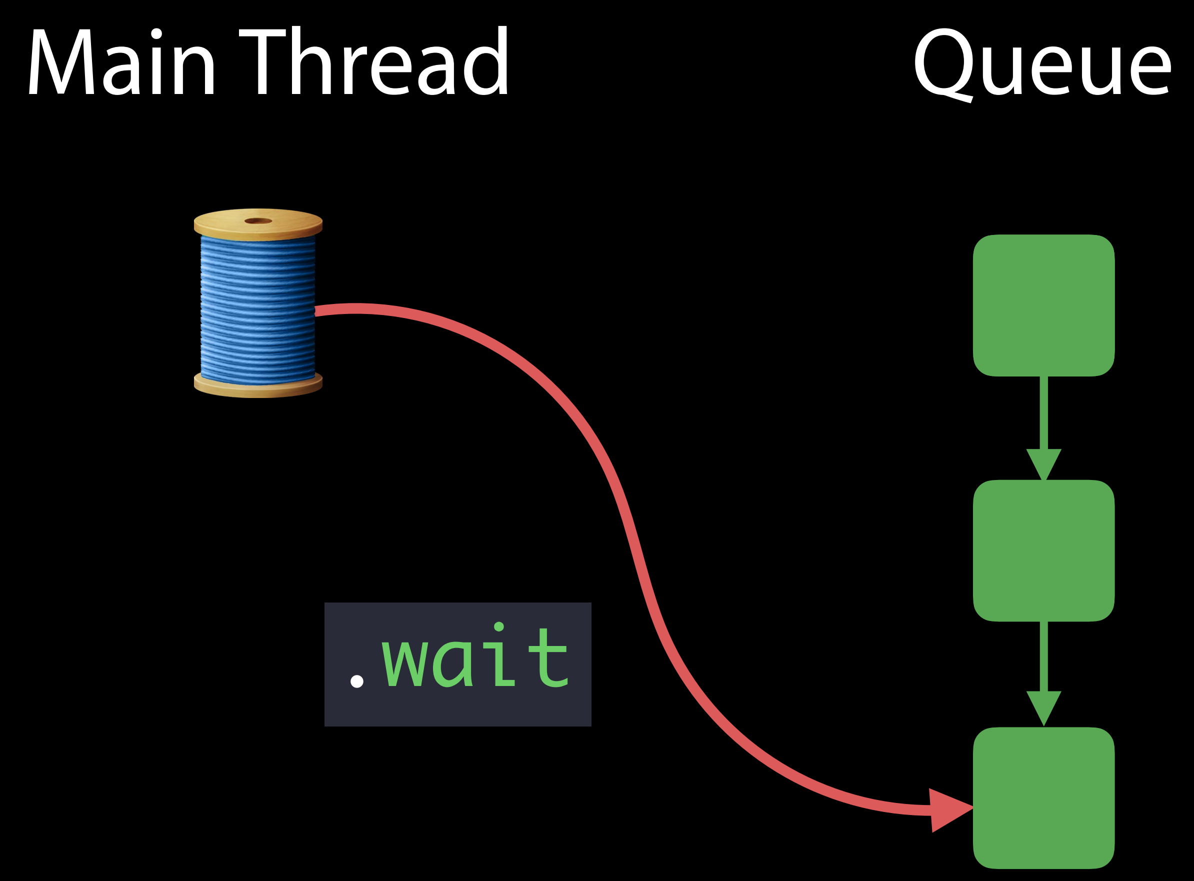
Use `.wait` on work items to signal that this item needs to execute



Waiting for Work Items

Use `.wait` on work items to signal that this item needs to execute

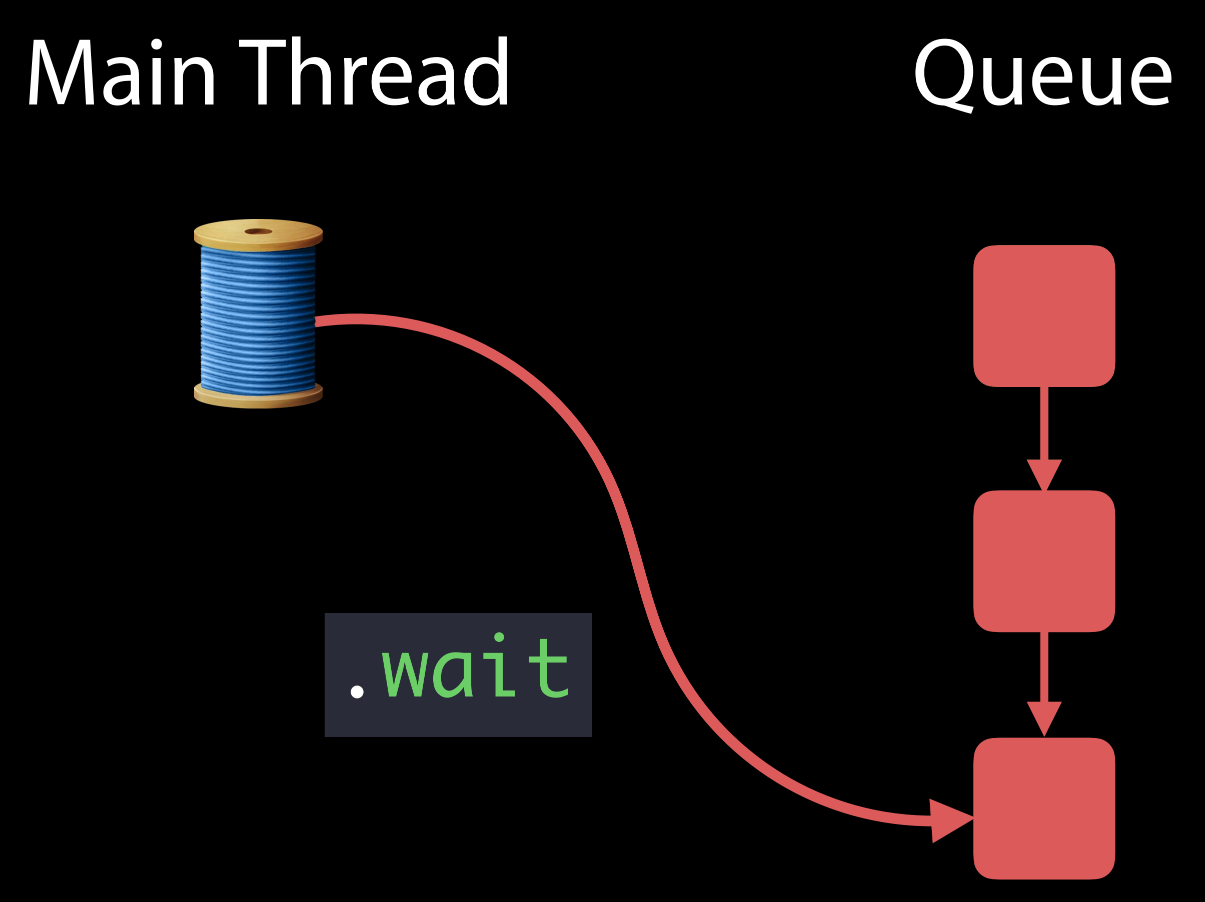
Dispatch elevates priority of queued work ahead



Waiting for Work Items

Use `.wait` on work items to signal that this item needs to execute

Dispatch elevates priority of queued work ahead

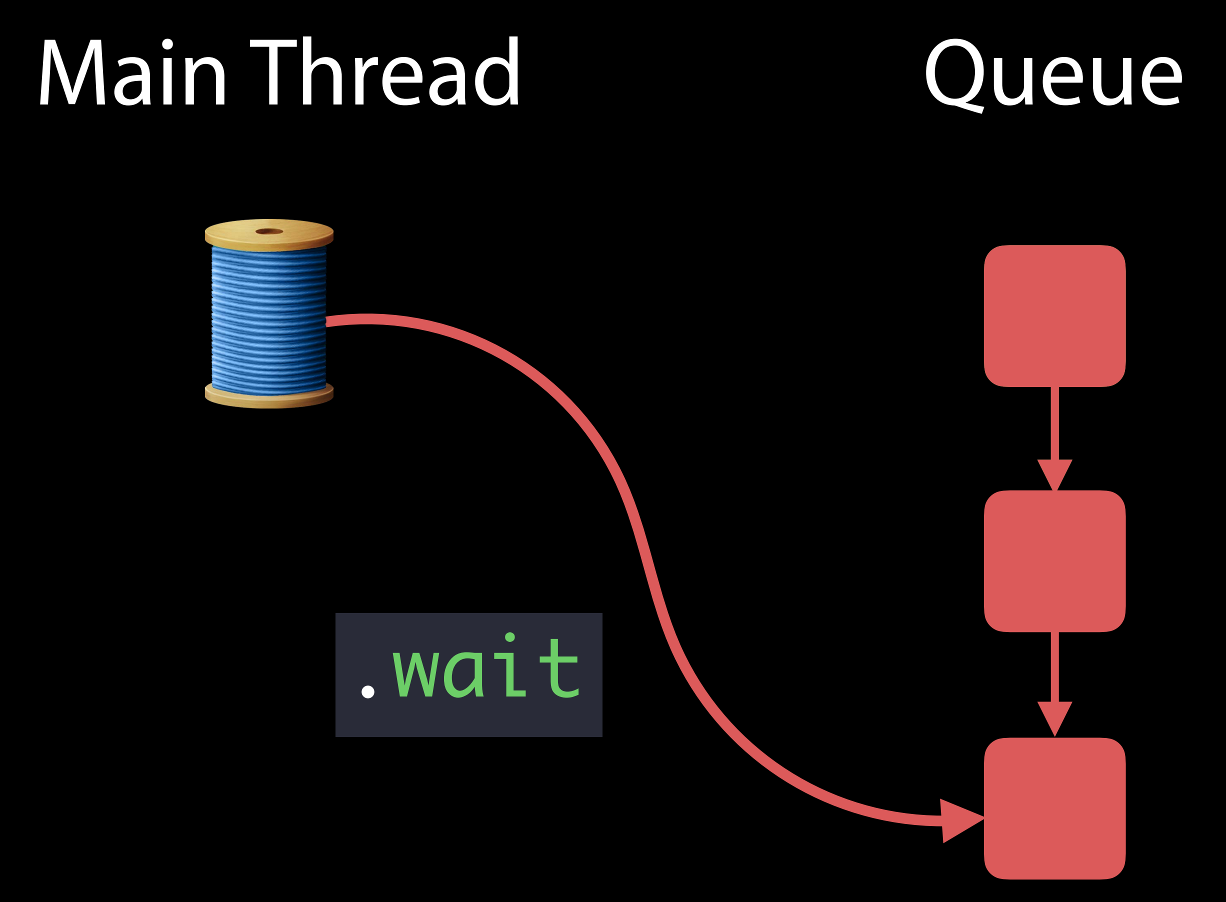


Waiting for Work Items

Use `.wait` on work items to signal that this item needs to execute

Dispatch elevates priority of queued work ahead

Waiting with a `DispatchWorkItem` gives ownership information



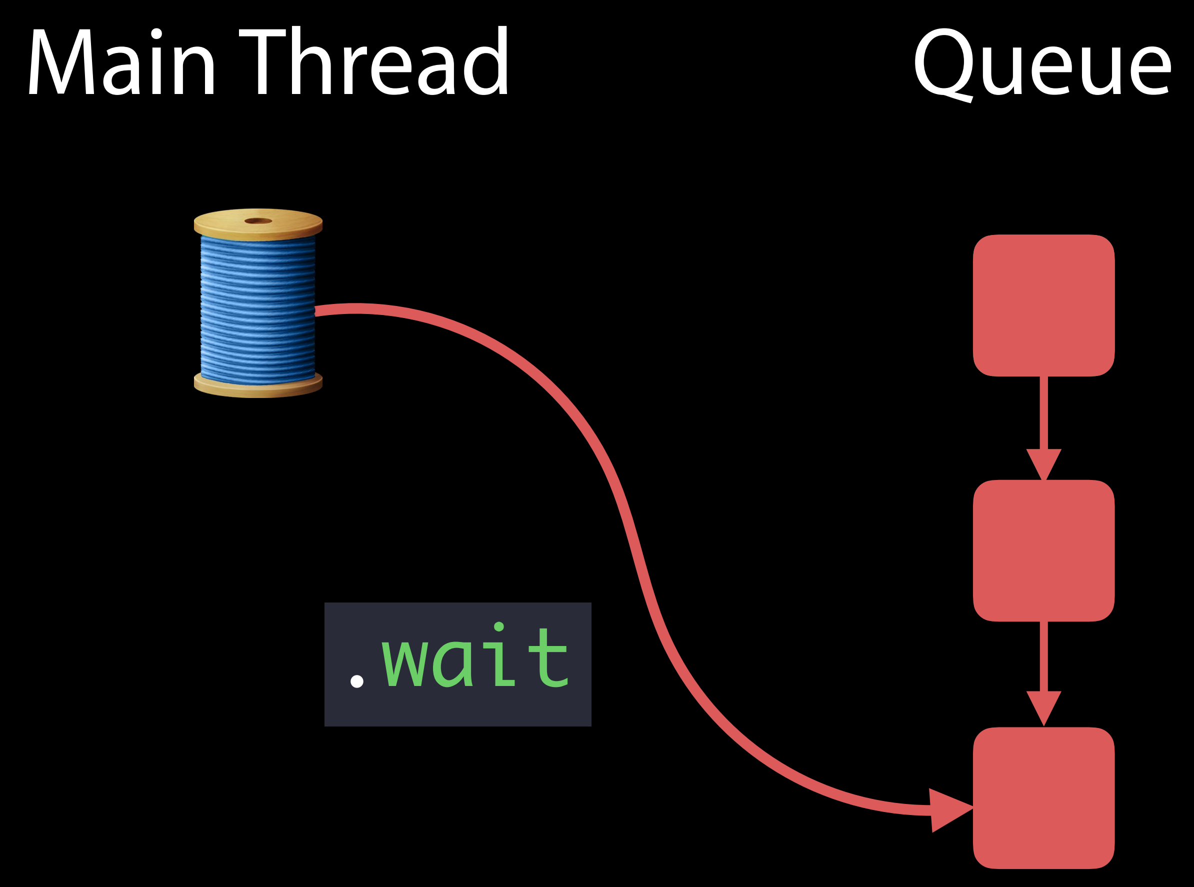
Waiting for Work Items

Use `.wait` on work items to signal that this item needs to execute

Dispatch elevates priority of queued work ahead

Waiting with a `DispatchWorkItem` gives ownership information

Semaphores and Groups do not admit a concept of ownership



Shared State Synchronization

Pierre Habouzit Darwin Runtime Engineer

Swift 3 and Synchronization

Synchronization is not part of the language in Swift 3

Global variables are initialized atomically

Swift 3 and Synchronization

Synchronization is not part of the language in Swift 3

Global variables are initialized atomically

Class properties are not atomic

Swift 3 and Synchronization

Synchronization is not part of the language in Swift 3

Global variables are initialized atomically

Class properties are not atomic

Lazy properties are not initialized atomically

> “There is no such thing as a benign race.”

Herb Sutter Chair of the ISO C++ standards committee

> “There is no such thing as a benign race.”

Herb Sutter Chair of the ISO C++ standards committee

Thread Sanitizer and Static Analysis	Mission	Thursday 10:00AM

Traditional C Locks in Swift

The Darwin module exposes traditional C lock types

- correct use of C struct based locks such as `pthread_mutex_t` is incredibly hard

Correct Use of Traditional Locks

`Foundation.Lock` can be used safely because it is a class

Correct Use of Traditional Locks

`Foundation.Lock` can be used safely because it is a class

Derive an Objective-C base class with struct based locks as ivars

```
@implementation LockableObject {
    os_unfair_lock _lock;
}

- (instancetype)init ...
- (void)lock   { os_unfair_lock_lock(&_lock); }
- (void)unlock { os_unfair_lock_unlock(&_lock); }
@end
```

Correct Use of Traditional Locks

`Foundation.Lock` can be used safely because it is a class

Derive an Objective-C base class with struct based locks as ivars

```
@implementation LockableObject {
   os_unfair_lock _lock;
}

- (instancetype)init ...
- (void)lock   { os_unfair_lock_lock(&_lock); }
- (void)unlock { os_unfair_lock_unlock(&_lock); }
@end
```

Use GCD for Synchronization

Use `DispatchQueue.sync(execute:)`

- harder to misuse than traditional locks, more robust
- better instrumentation (Xcode, assertions, …)

```
// Use Explicit Synchronization

class MyObject {
    private let internalState: Int
    private let internalQueue: DispatchQueue









}
```

```swift
// Use Explicit Synchronization

class MyObject {
   private let internalState: Int
   private let internalQueue: DispatchQueue
   var state: Int {
      get {
         return internalQueue.sync { internalState }
      }



   }
}
```

✓

```swift
// Use Explicit Synchronization

class MyObject {
   private let internalState: Int
   private let internalQueue: DispatchQueue
   var state: Int {
      get {
         return internalQueue.sync { internalState }
      }
      set (newState) {
         internalQueue.sync { internalState = newState }
      }
   }
}
```

Preconditions

Avoid data corruption

NEW

GCD lets you express several preconditions

Preconditions

Avoid data corruption

NEW

GCD lets you express several preconditions

- Code is running on a given queue

```
dispatchPrecondition(.onQueue(expectedQueue))
```

Preconditions

Avoid data corruption

NEW

GCD lets you express several preconditions

- Code is running on a given queue
- Code is not running on a given queue

```
dispatchPrecondition(.onQueue(expectedQueue))

dispatchPrecondition(.notOnQueue(unexpectedQueue))
```

Object Lifecycle in a Concurrent World

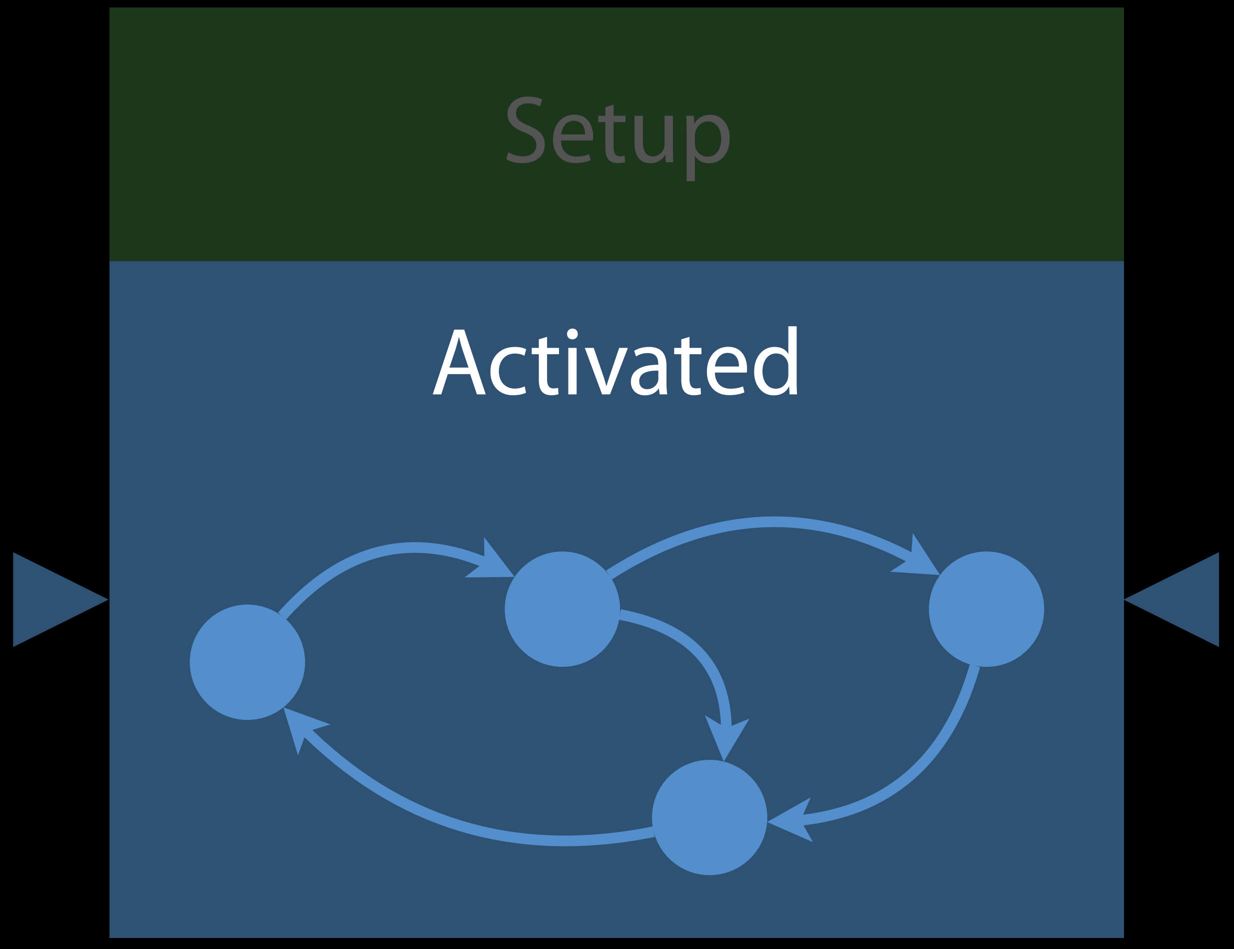
Object Lifecycle in a Concurrent World

Object Lifecycle in a Concurrent World

1. Single threaded setup

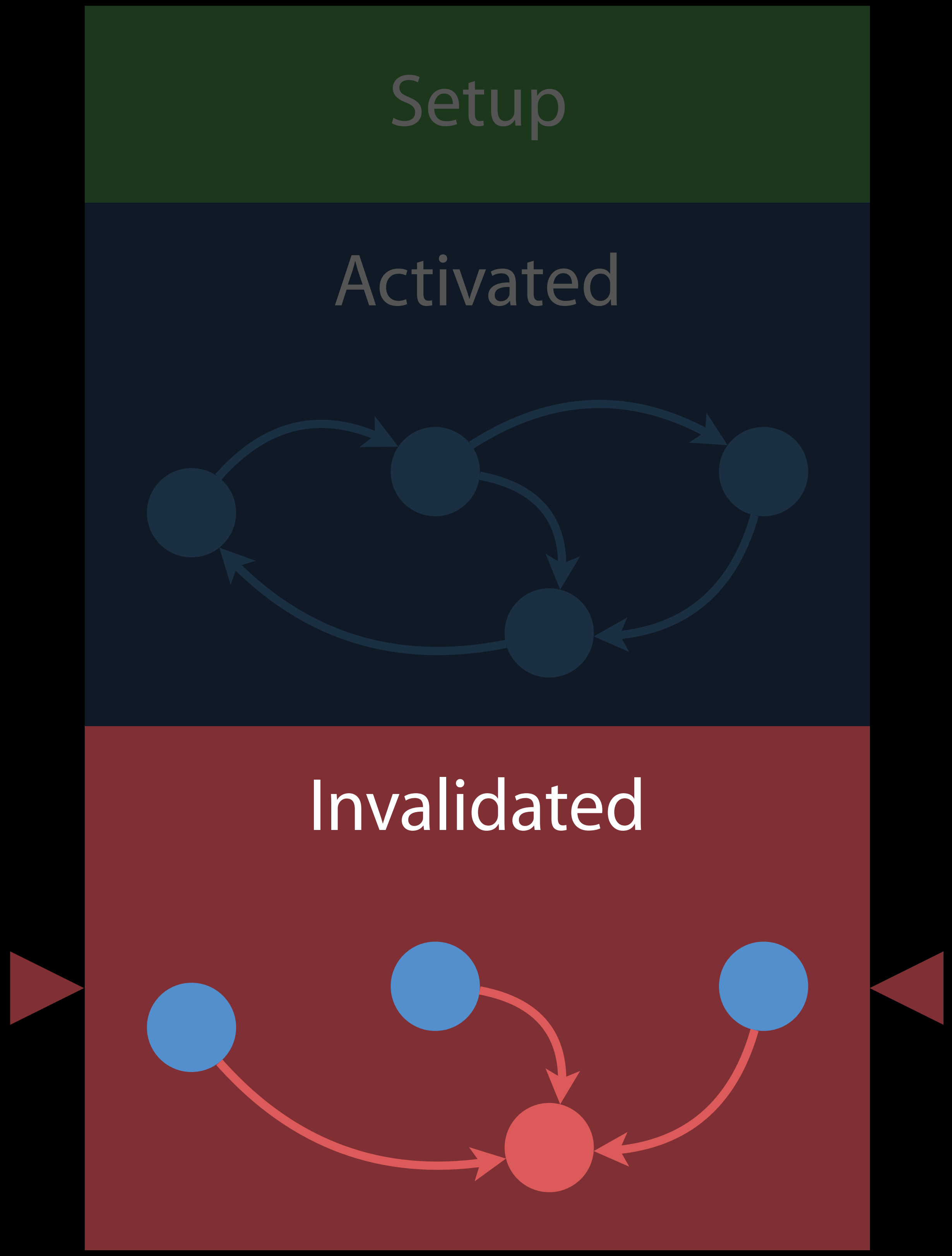
Object Lifecycle in a Concurrent World

1. Single threaded setup
2. `activate` the concurrent state machine



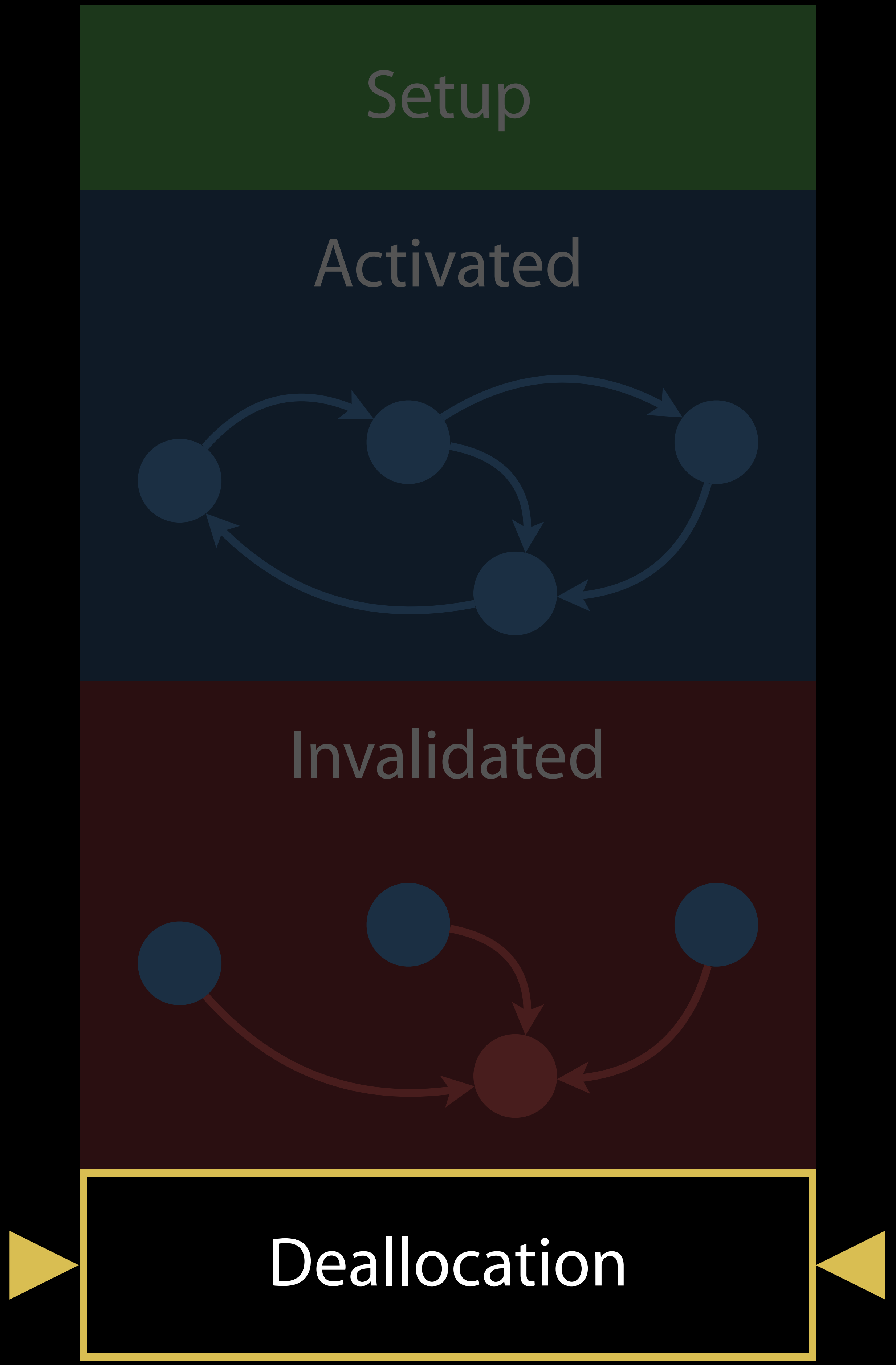
Object Lifecycle in a Concurrent World

1. Single threaded setup
2. `activate` the concurrent state machine
3. `invalidate` the concurrent state machine



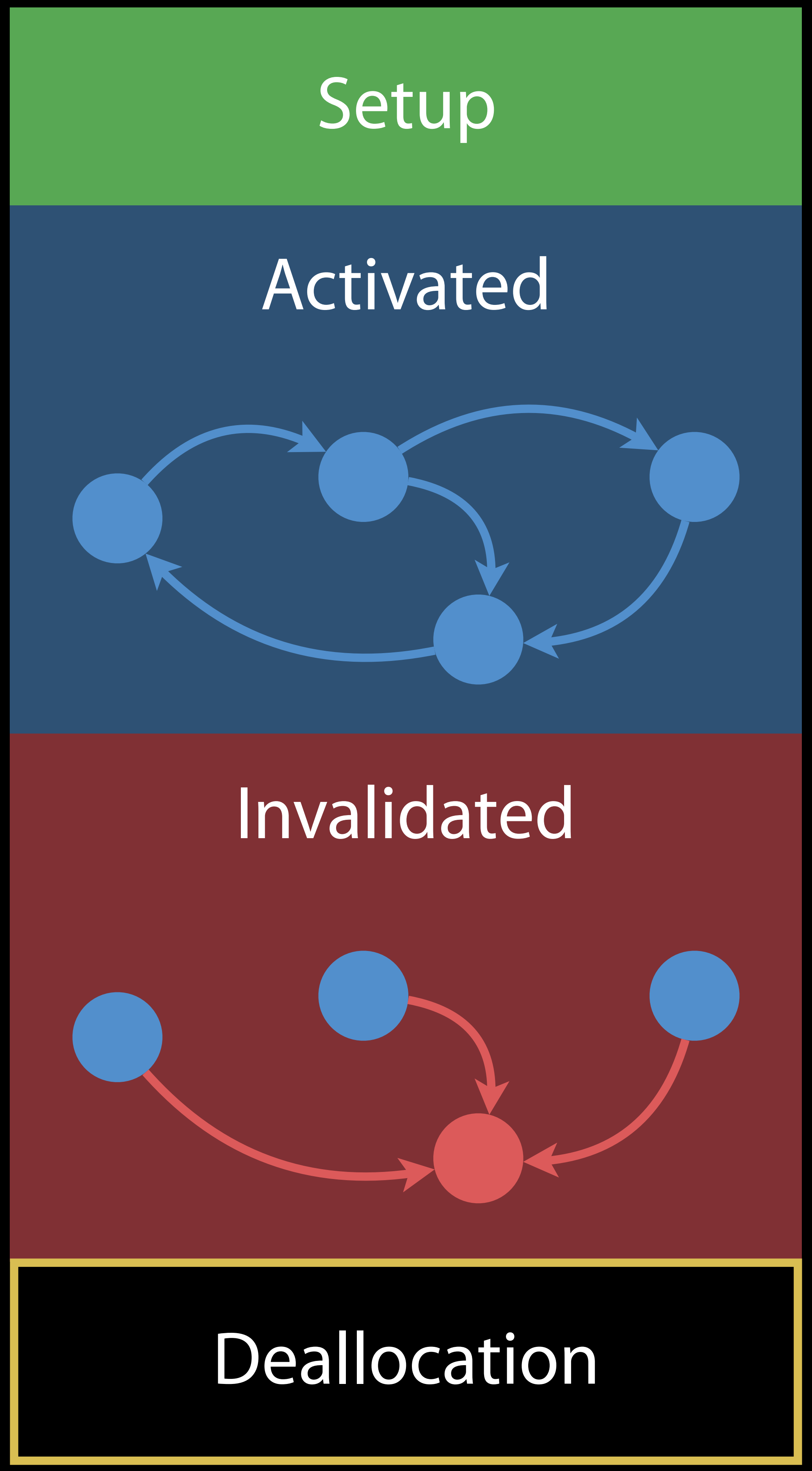
Object Lifecycle in a Concurrent World

1. Single threaded setup
2. `activate` the concurrent state machine
3. `invalidate` the concurrent state machine
4. Single threaded deallocation

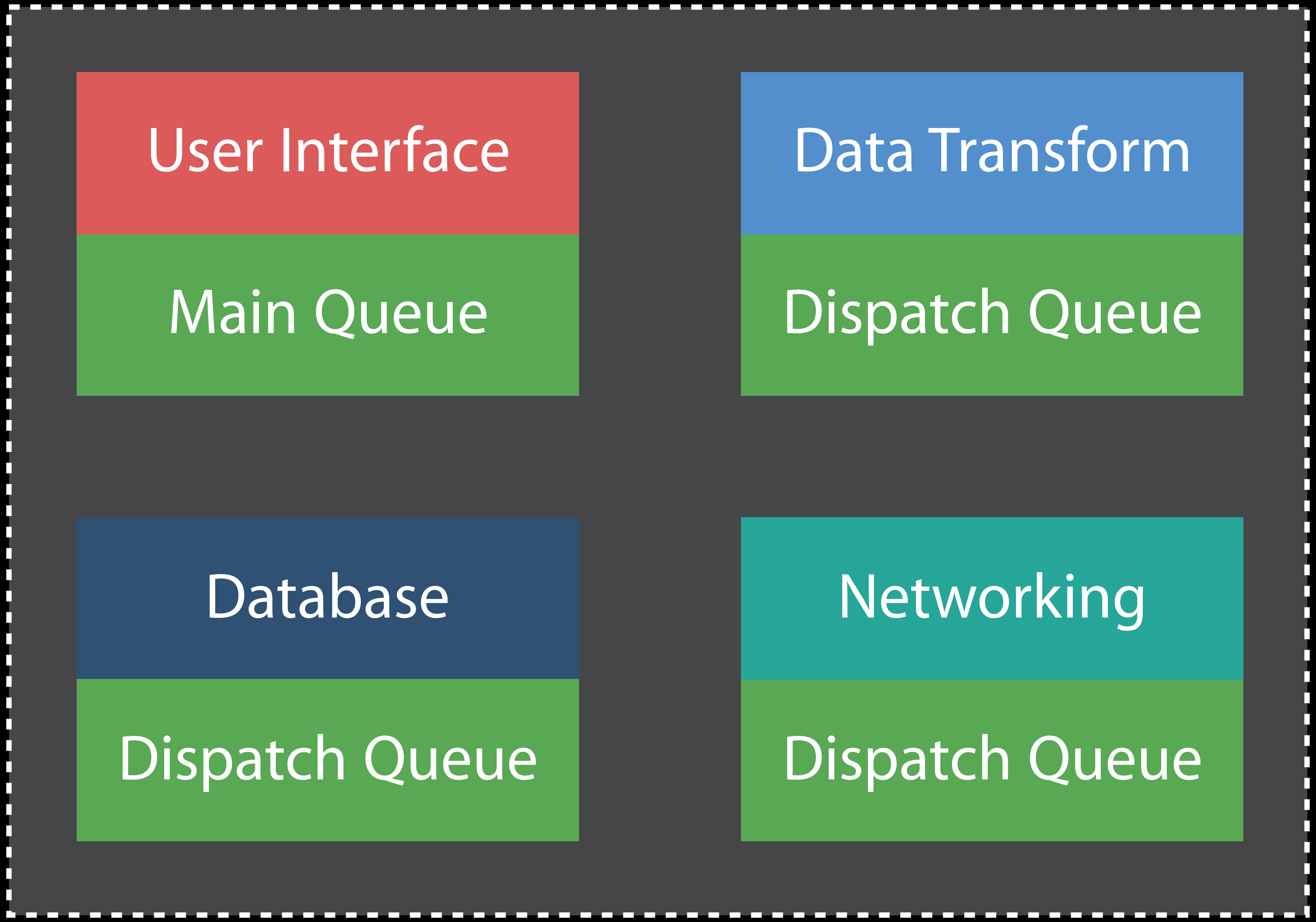


Object Lifecycle in a Concurrent World

1. Single threaded setup
2. `activate` the concurrent state machine
3. `invalidate` the concurrent state machine
4. Single threaded deallocation



Observer Pattern



User Interface

Main Queue

Data Transform

Dispatch Queue

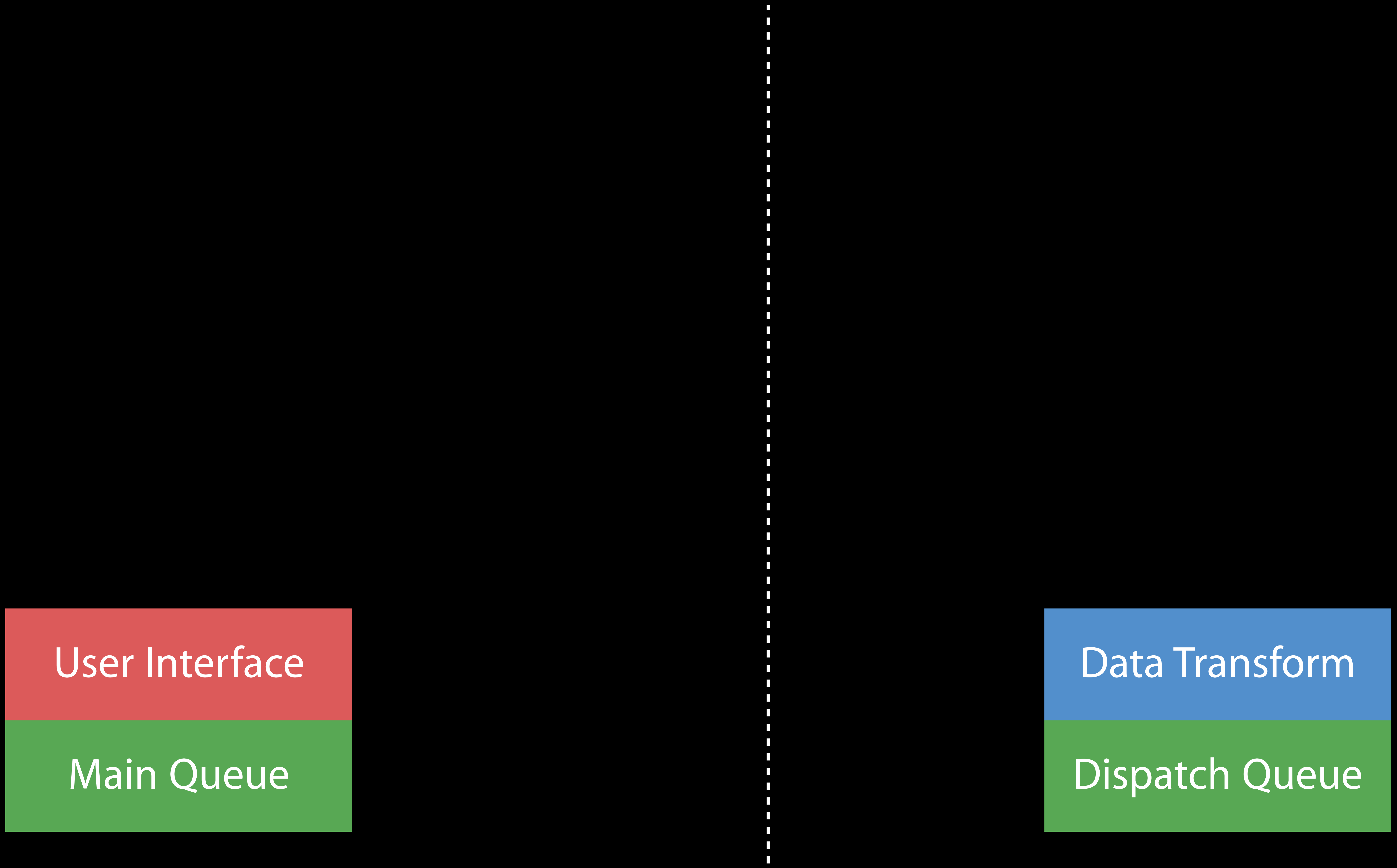
Database

Dispatch Queue

Networking

Dispatch Queue

Observer Pattern

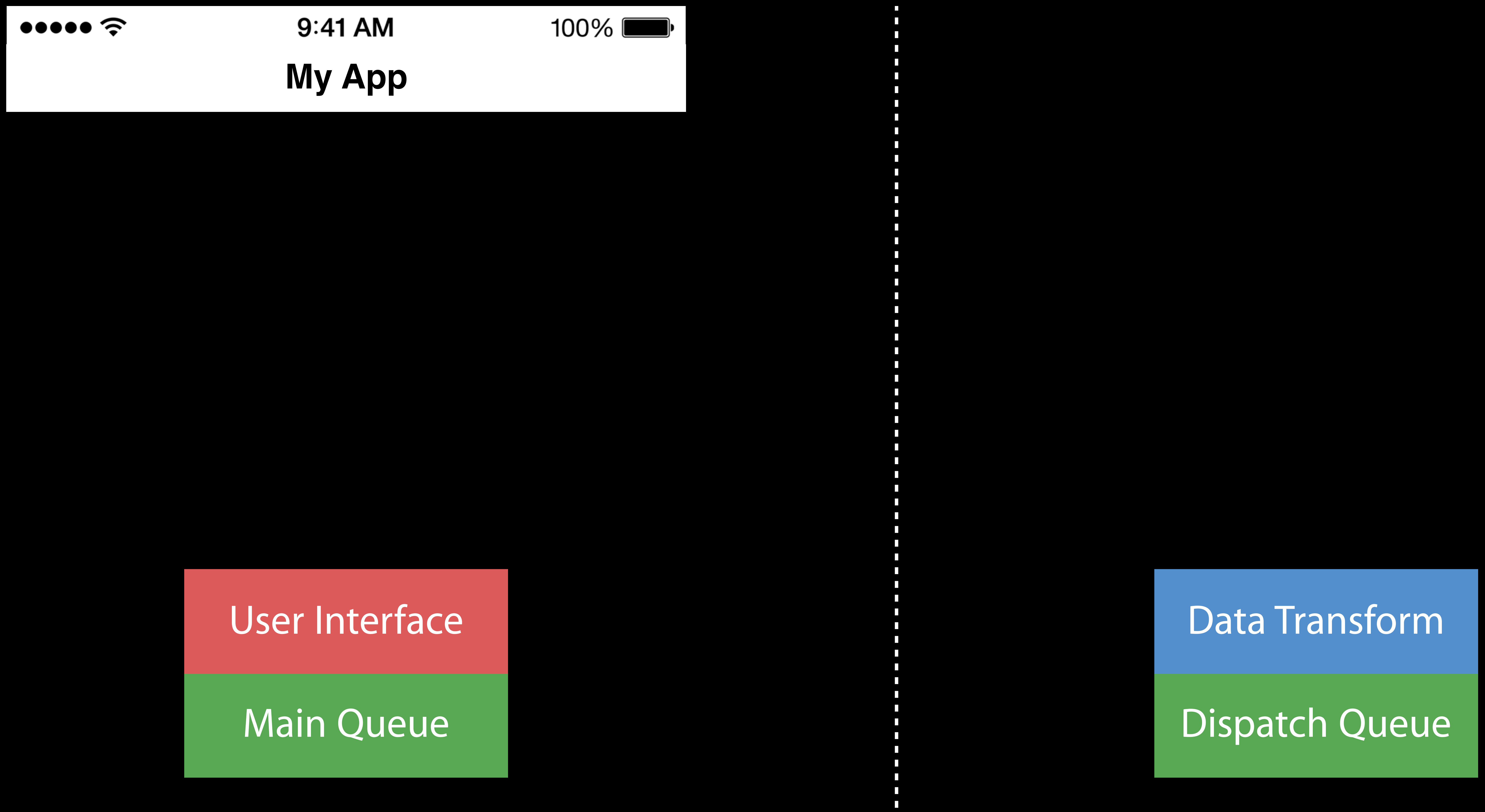

User Interface

Main Queue

Data Transform

Dispatch Queue

Observer Pattern



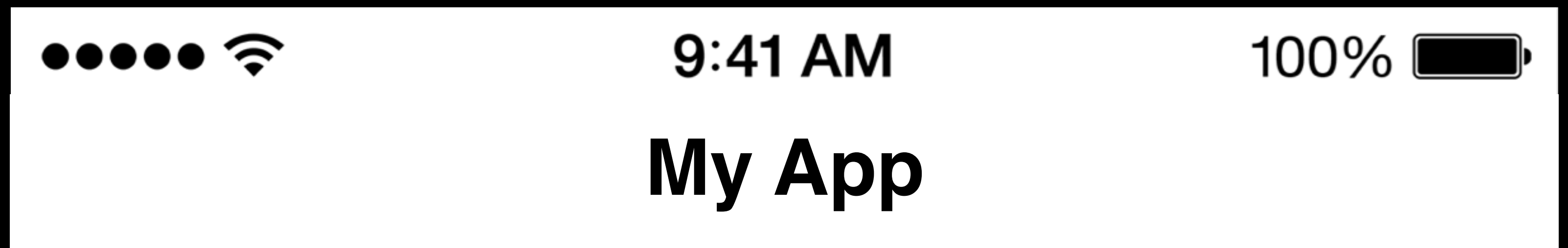
User Interface

Main Queue

Data Transform

Dispatch Queue

Observer Pattern



```
class BusyController: SubsystemObserving {
    // ...
}
```

User Interface

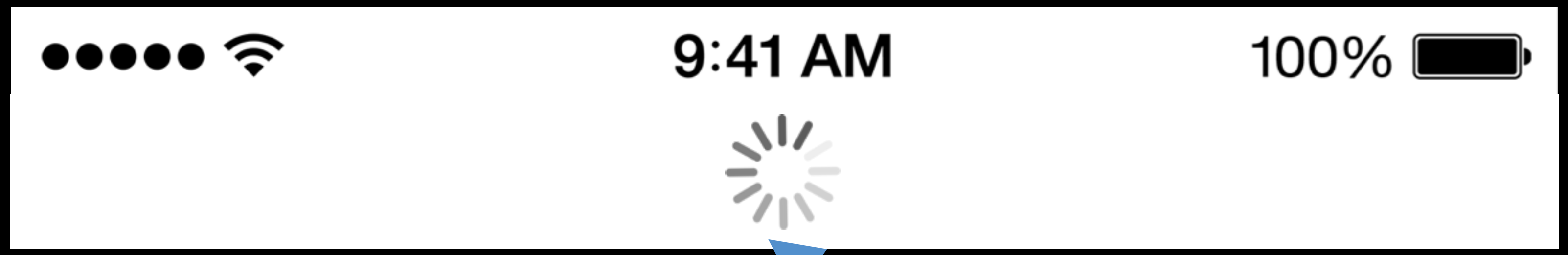
Main Queue

```
protocol SubsystemObserving {
    func systemStarted(...)
    func systemDone(...)
}
```

Data Transform

Dispatch Queue

Observer Pattern



```
class BusyController: SubsystemObserving {
    // ...
}
```

```
protocol SubsystemObserving {
    func systemStarted(...)
    func systemDone(...)
}
```

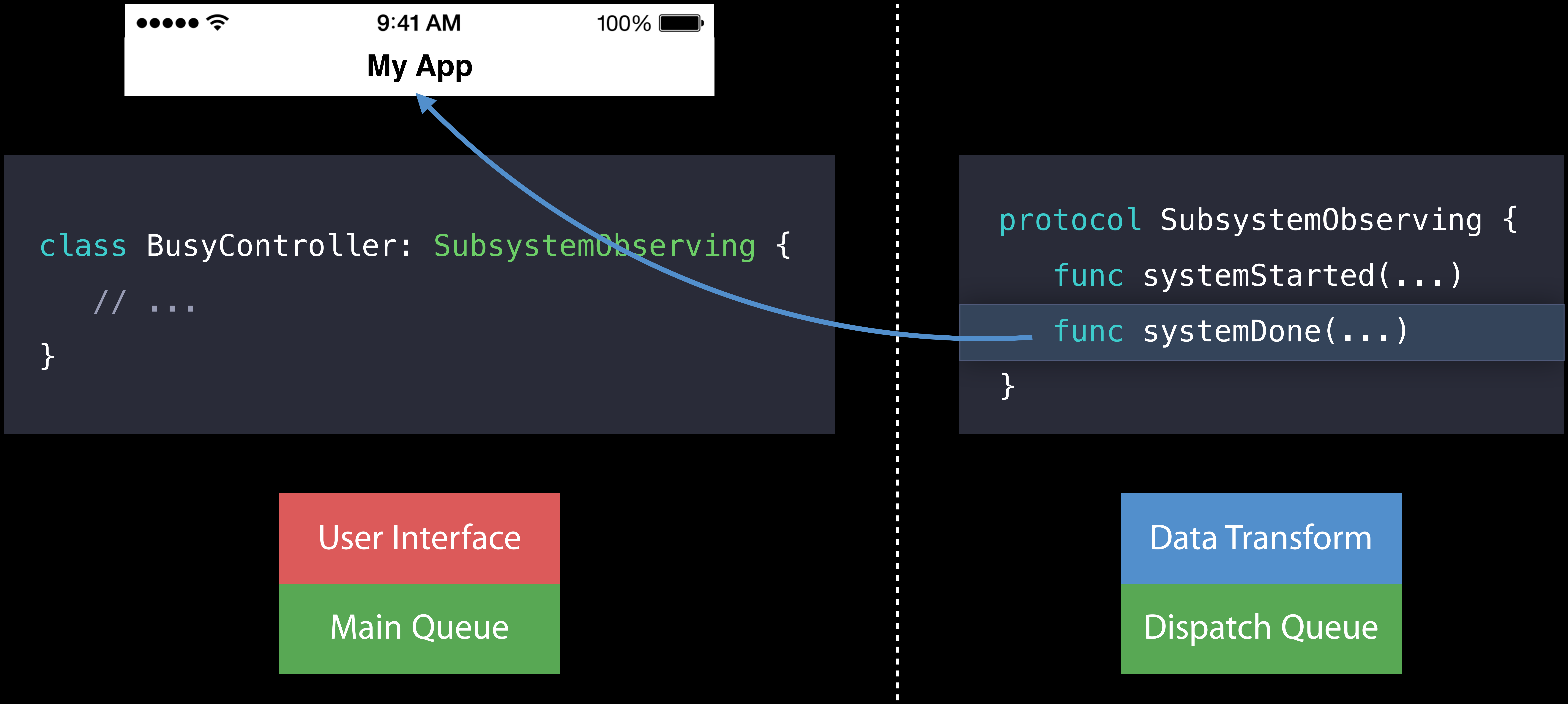
User Interface

Main Queue

Data Transform

Dispatch Queue

Observer Pattern



```
class BusyController: SubsystemObserving {
    // ...
}
```

```
protocol SubsystemObserving {
    func systemStarted(...)
    func systemDone(...)
}
```

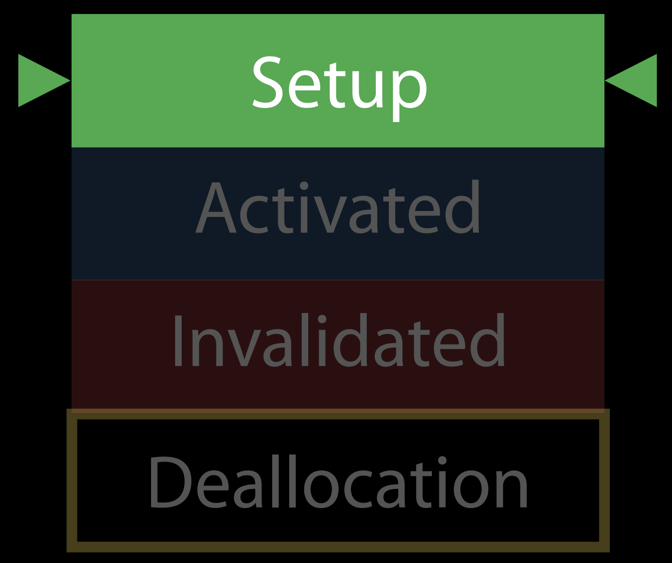
User Interface

Main Queue

Data Transform

Dispatch Queue

Setup



Setup

Activated

Invalidated

Deallocation

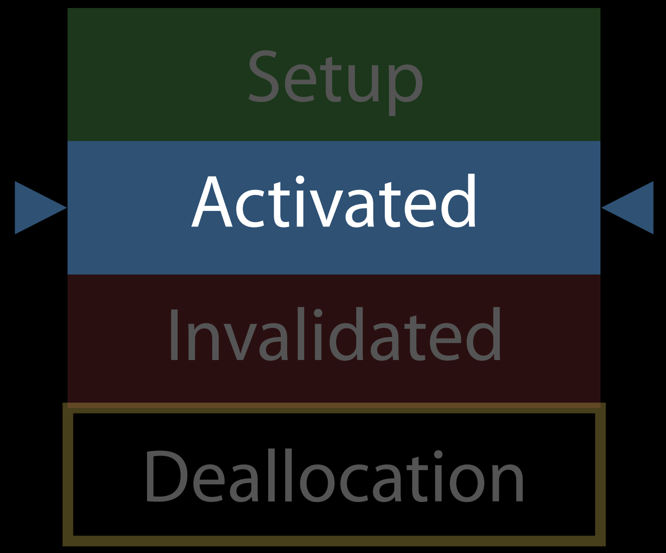
```
class BusyController: SubsystemObserving {
    init(...) { ... }




}
```

Activation

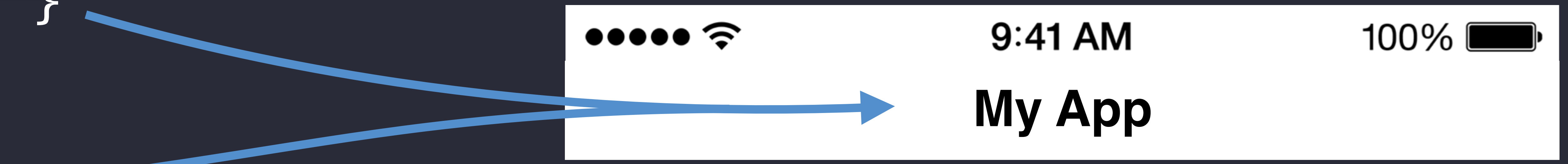
Setup
▶ Activated ◀
Invalidated
Deallocation

```
class BusyController: SubsystemObserving {
    init(...) { ... }

    func activate() {
        DataTransform.sharedInstance.register(observer: self, queue: DispatchQueue.main)
    }
}
```

Active State Machine



```
class BusyController: SubsystemObserving {
    func systemStarted(...) { /* ... */ }

    func systemDone(...) { /* ... */ }
}
```



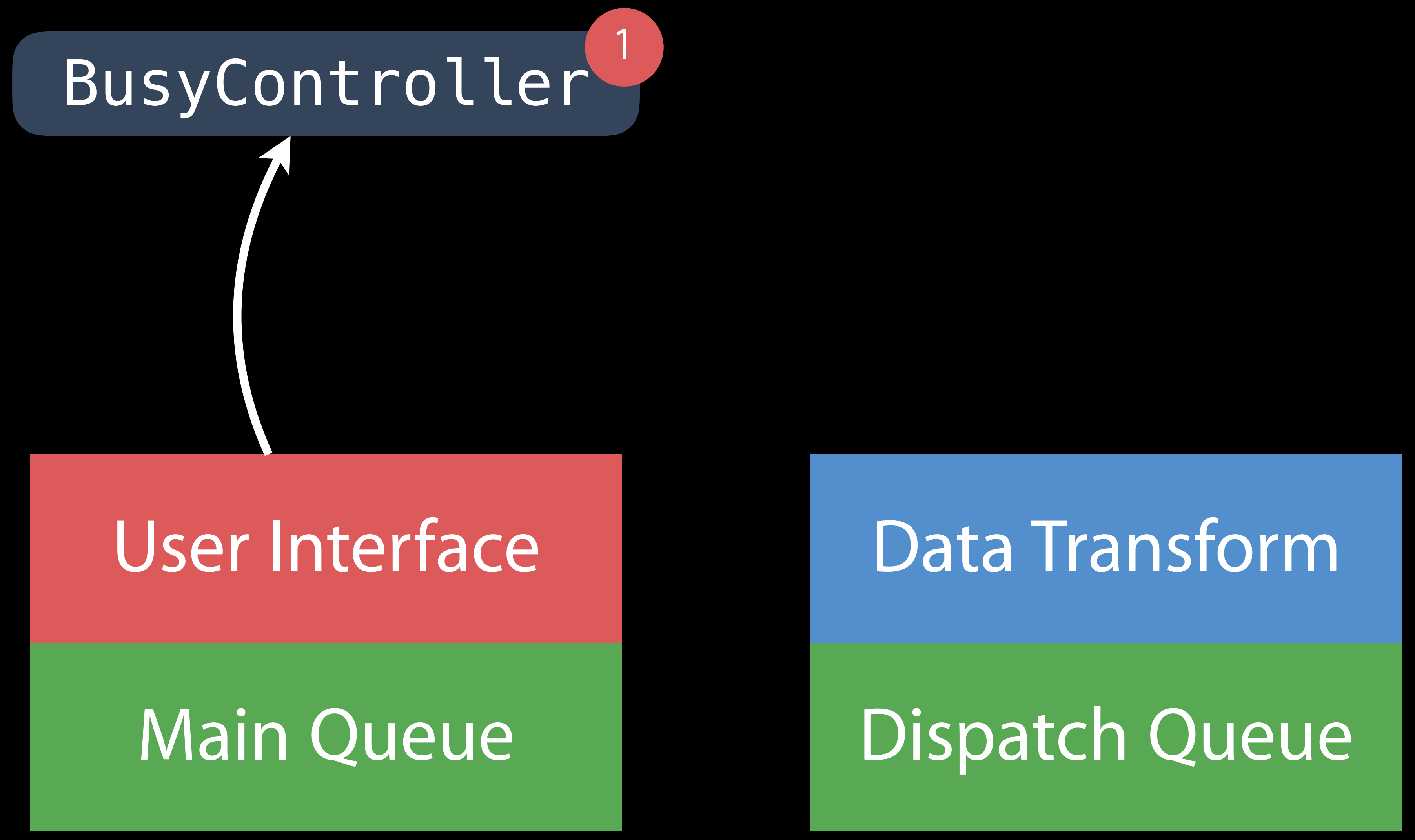
Deallocation

Setup
Activated
Invalidated
Deallocation

```
class BusyController: SubsystemObserving {
    deinit {
        DataTransform.sharedInstance.unregister(observer: self)
    }
}
```

Deallocation

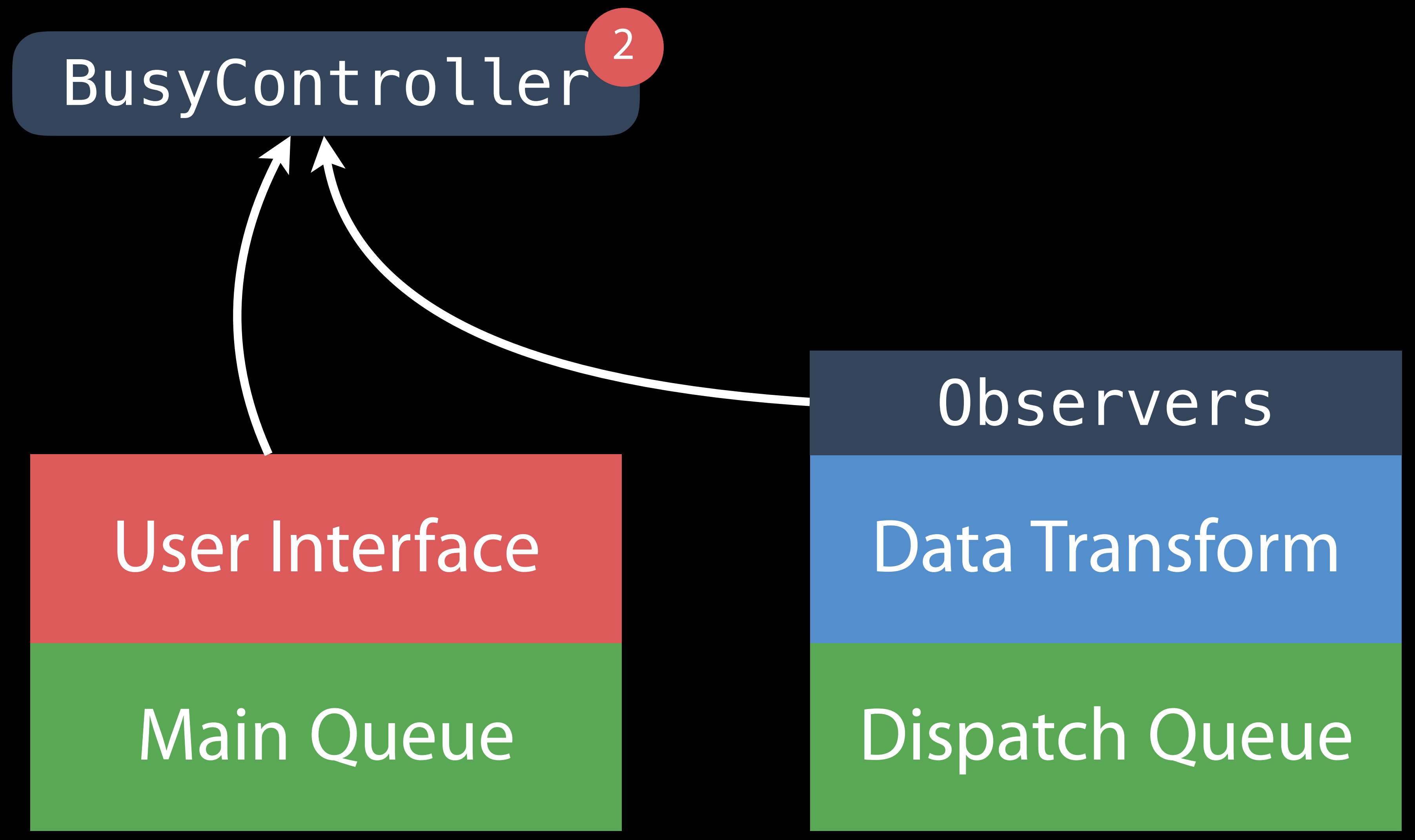
Setup
Activated
Invalidated
Deallocation



BusyController
1
User Interface
Main Queue
Data Transform
Dispatch Queue

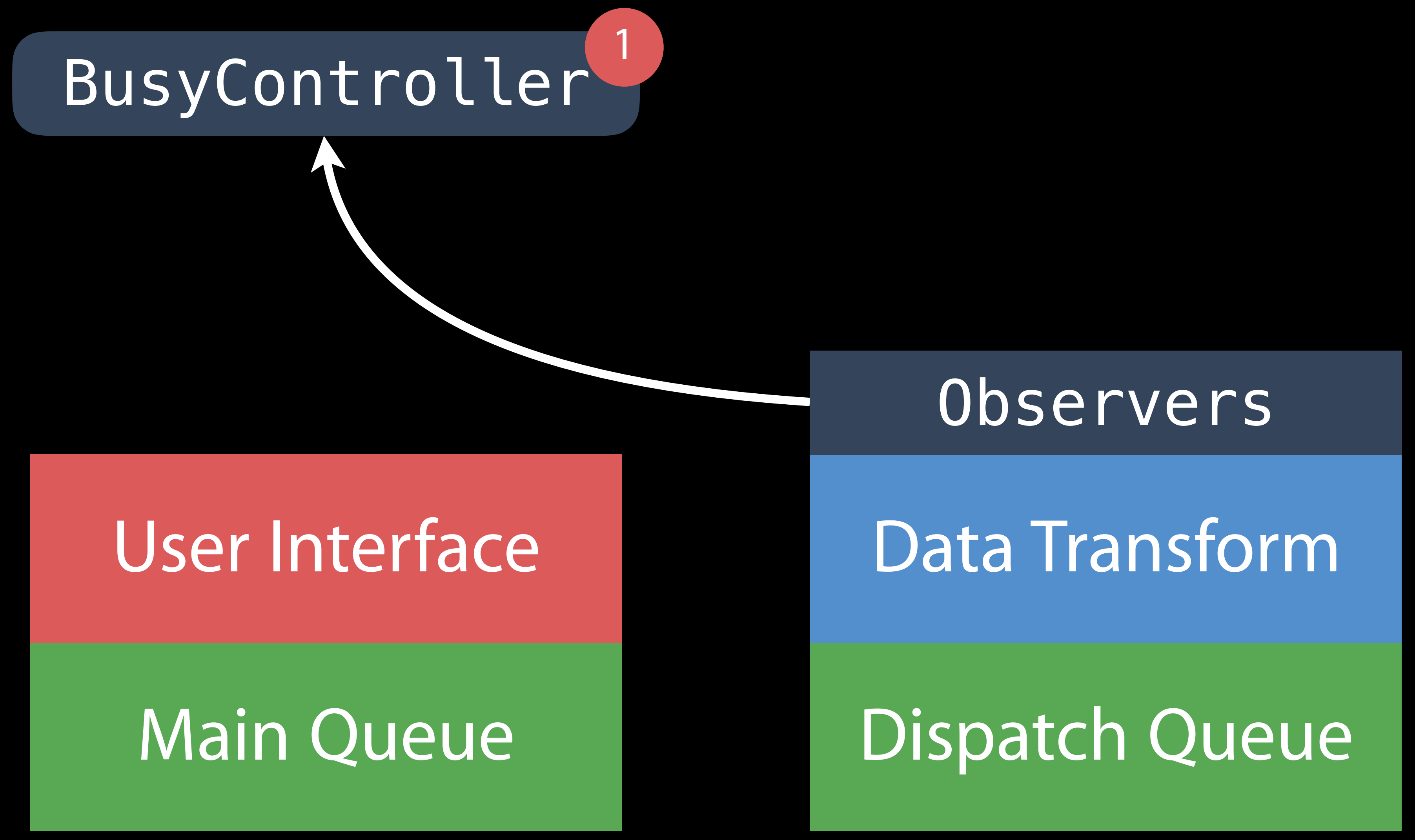
Deallocation

Setup
Activated
Invalidated
Deallocation



Deallocation

Setup
Activated
Invalidated
Deallocation



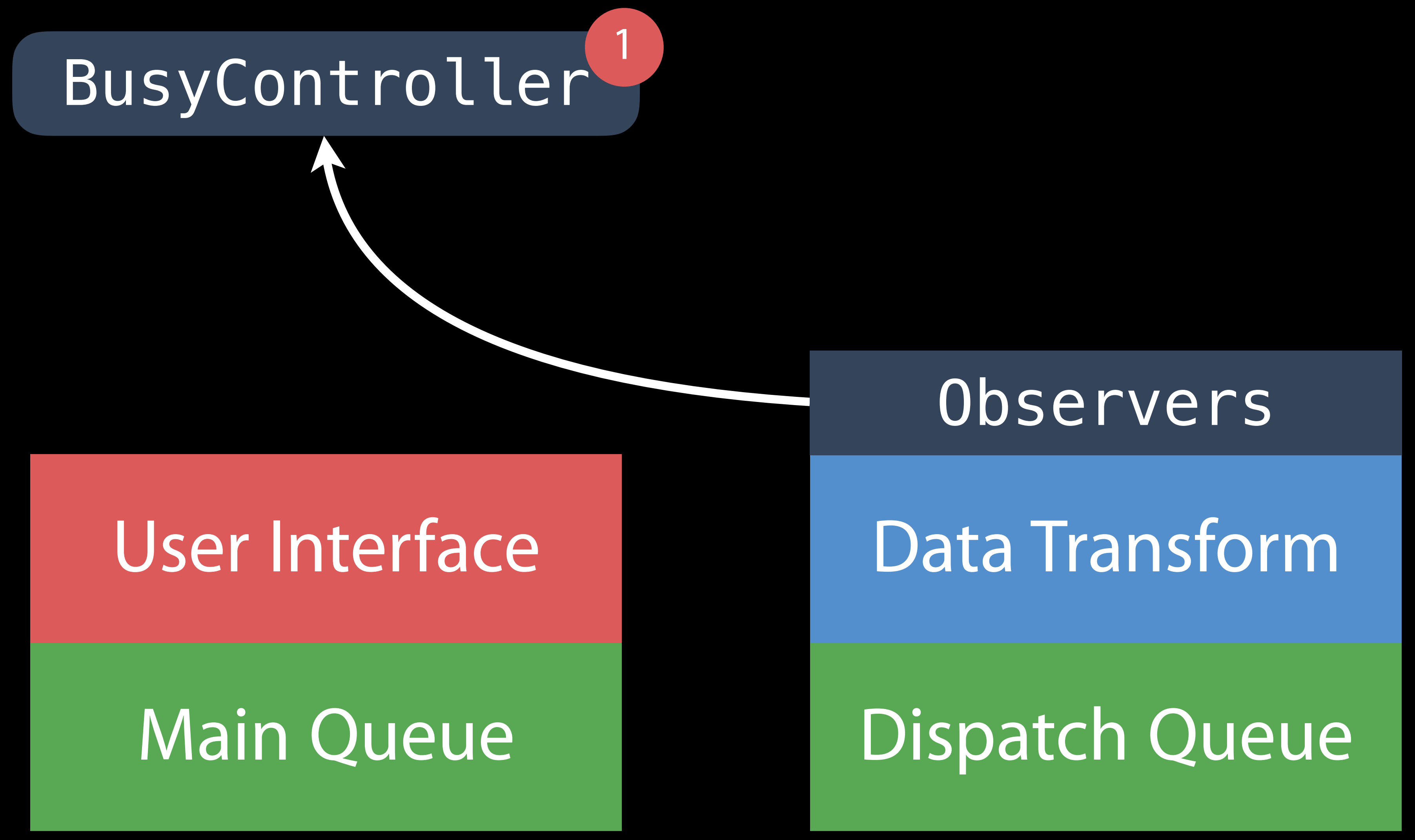
Deallocation

Setup

Activated

Invalidated

Deallocation


BusyController

1

Observers

User Interface

Main Queue

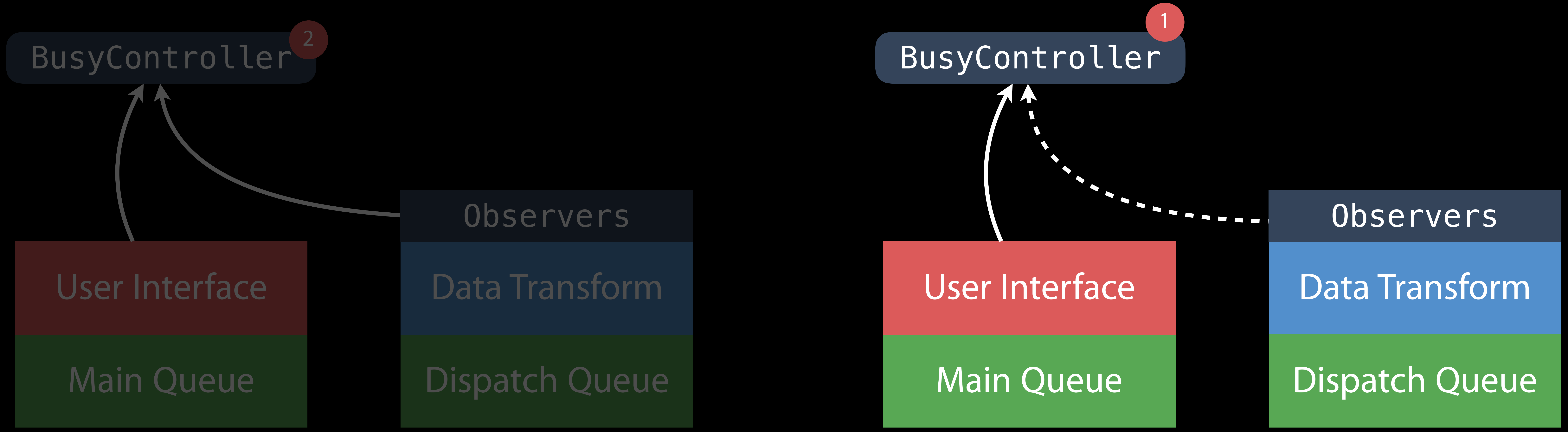
Data Transform

Dispatch Queue

✕ Abandoned memory

Deallocation

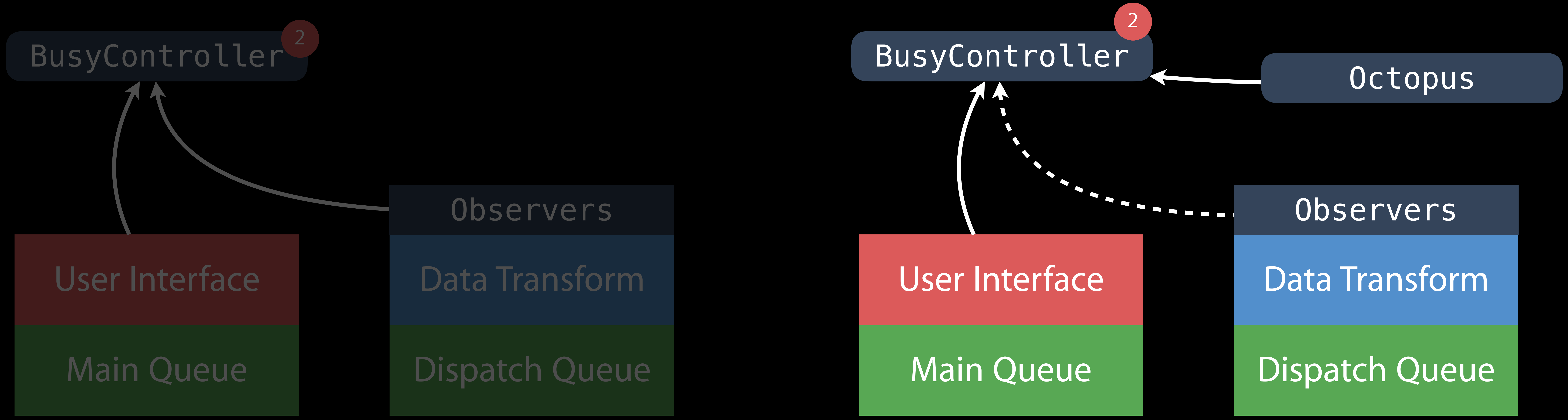
- Setup
- Activated
- Invalidated
- **Deallocation**



☒ Abandoned memory

Deallocation

Setup
Activated
Invalidated
Deallocation



✗ Abandoned memory

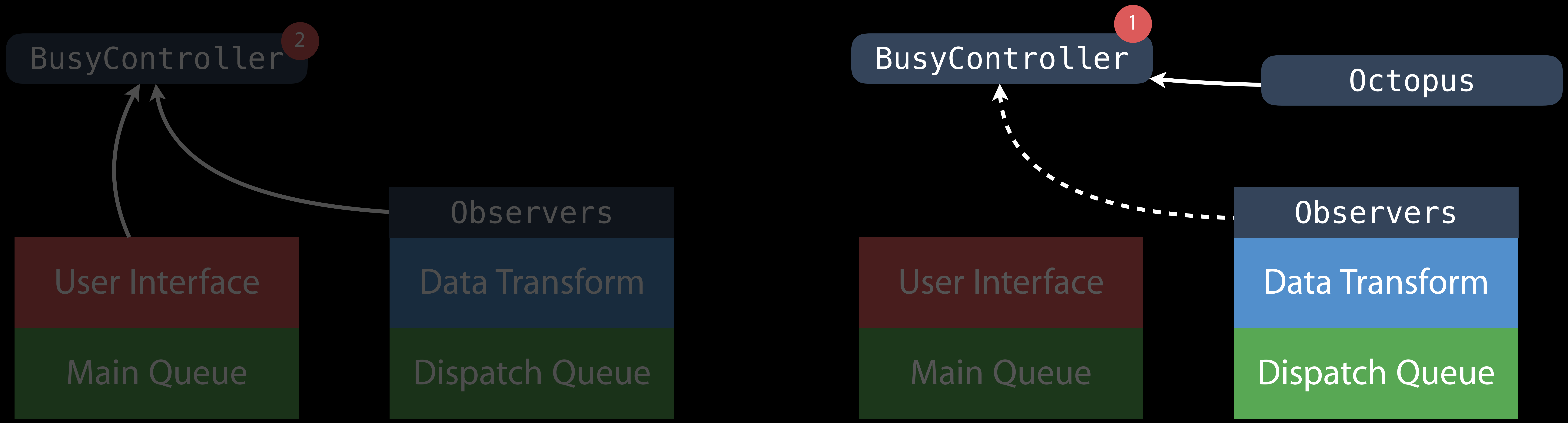
Deallocation

Setup

Activated

Invalidated

Deallocation



Abandoned memory

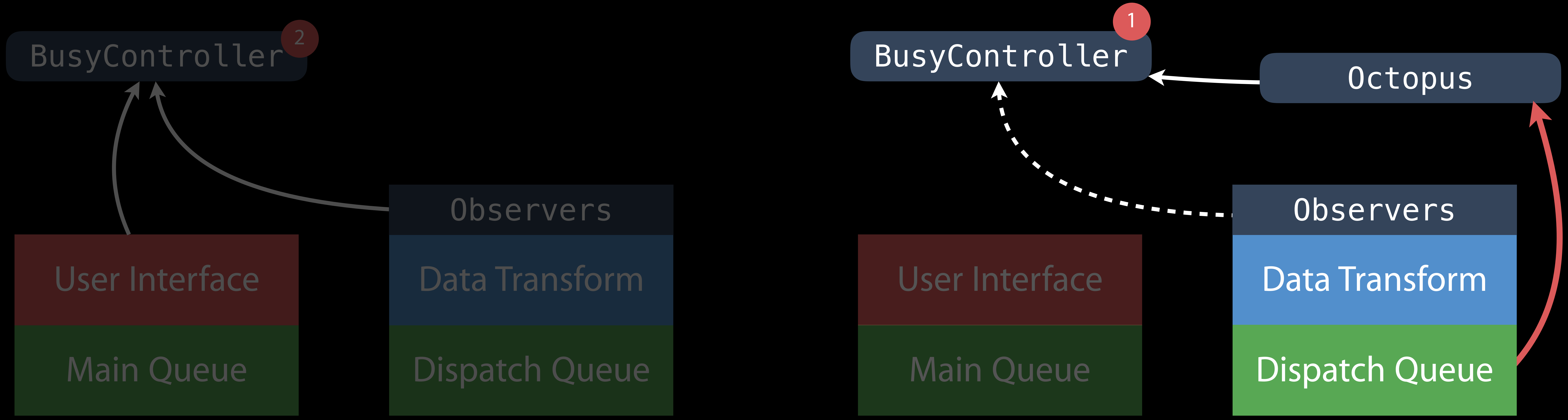
Deallocation

Setup

Activated

Invalidated

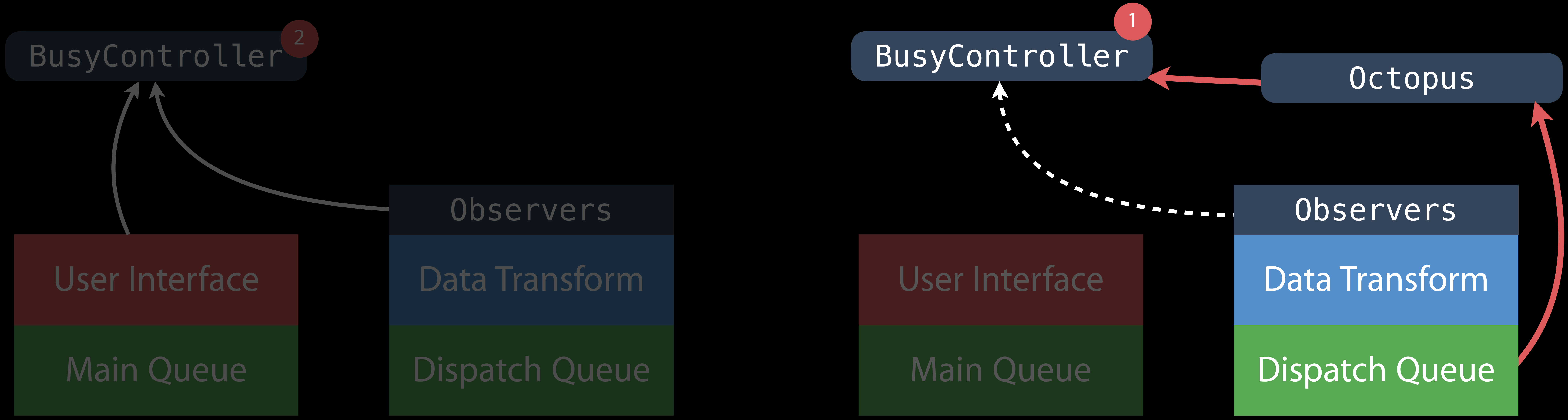
Deallocation



☒ Abandoned memory

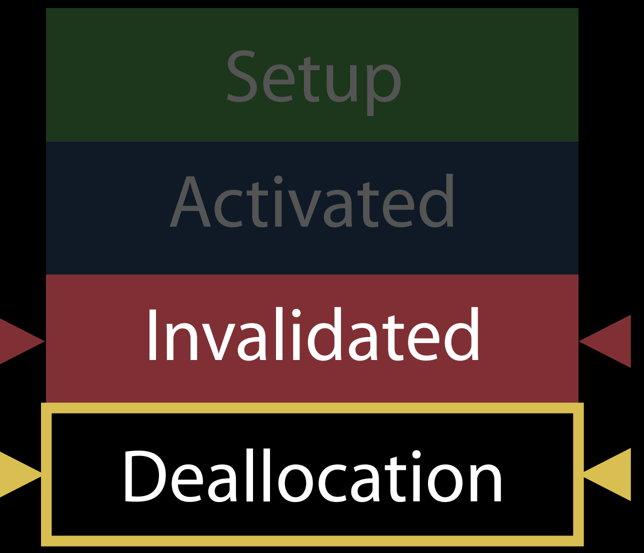
Deallocation

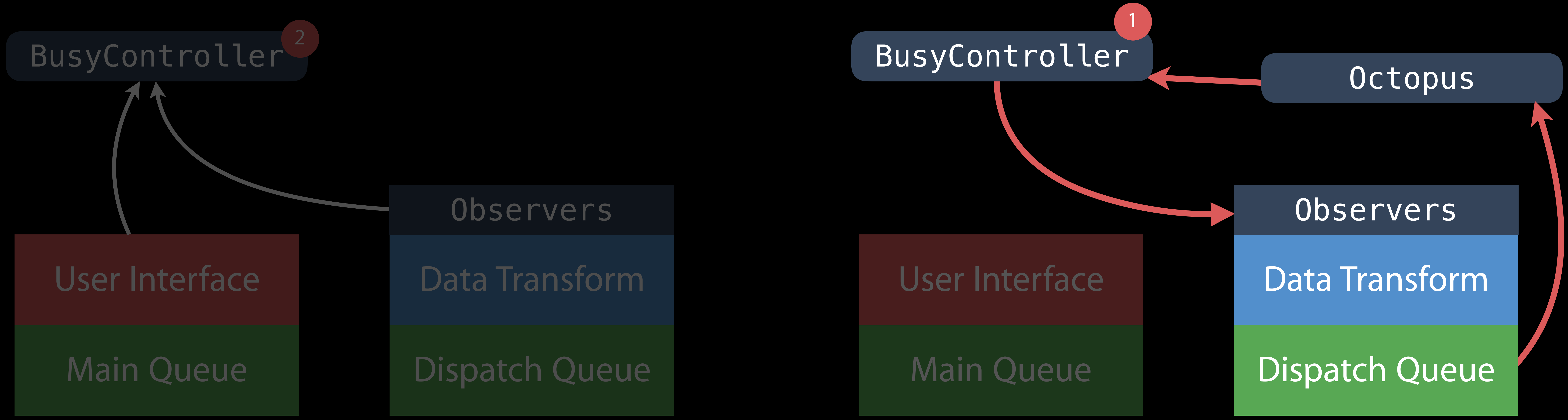
Setup
Activated
Invalidated
Deallocation



Abandoned memory

Deallocation

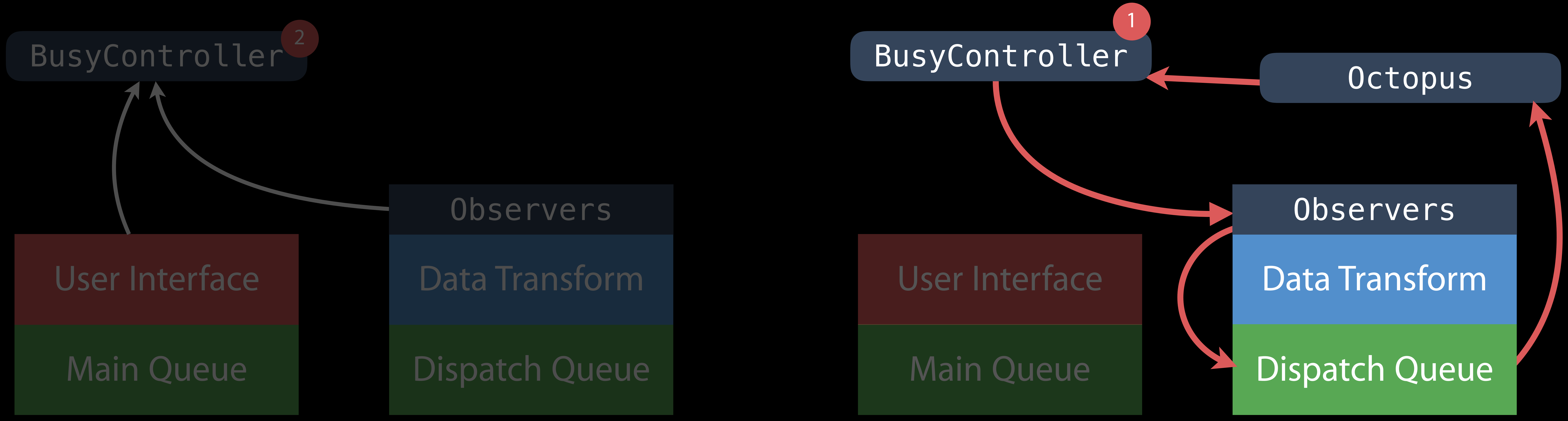




✕ Abandoned memory

Deallocation

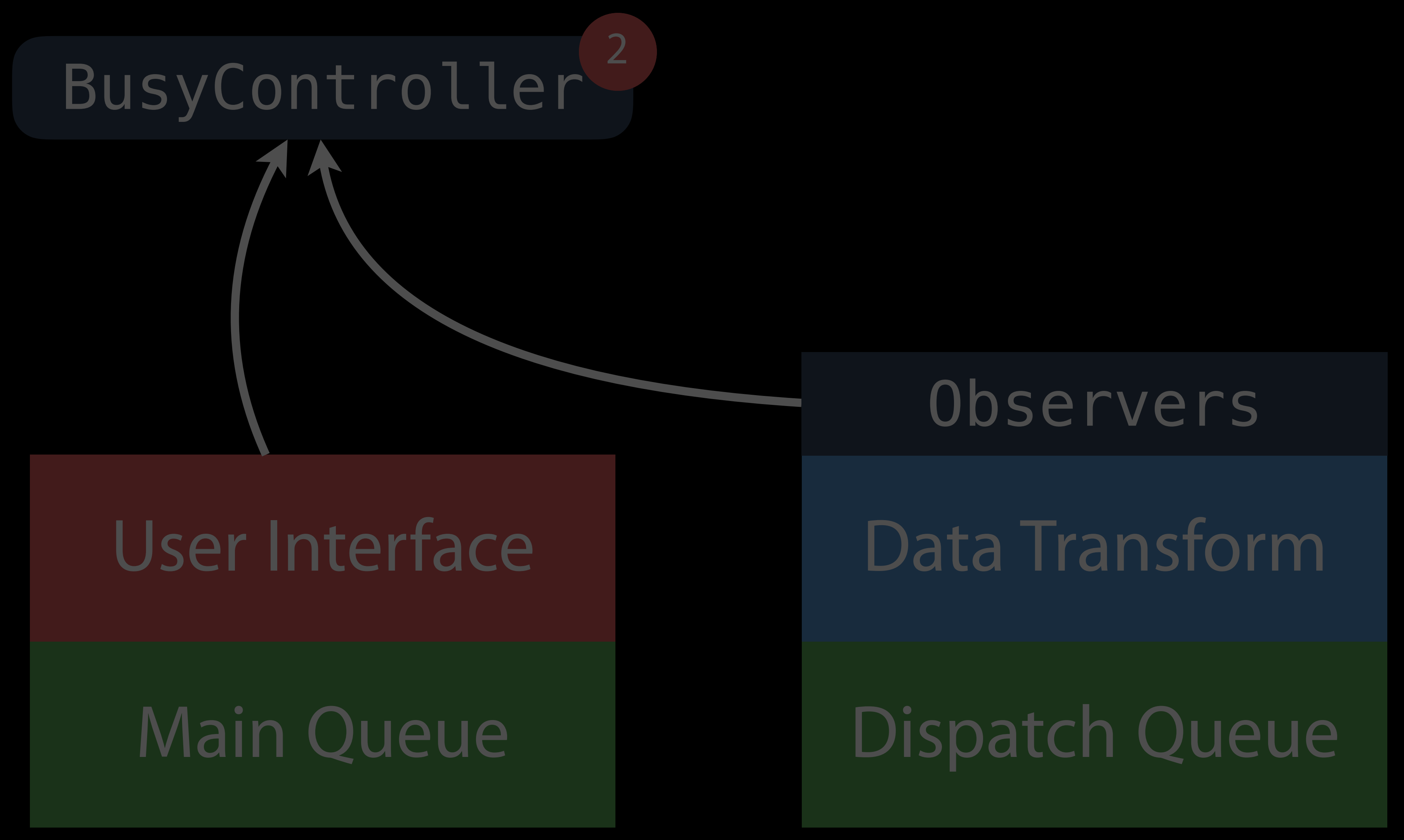
- Setup
- Activated
- Invalidated
- **Deallocation**

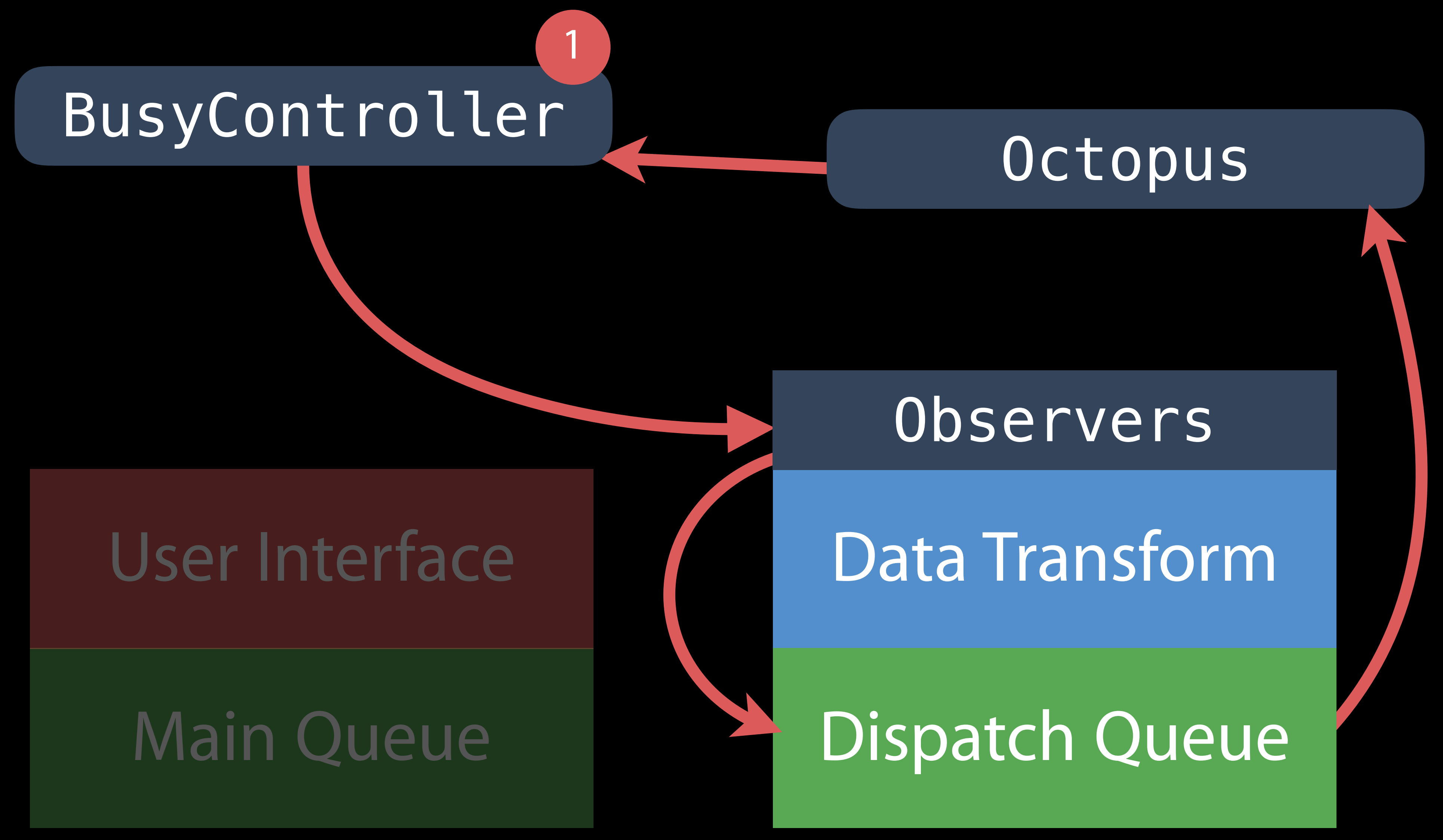


☒ Abandoned memory

Deallocation

Setup
Activated
Invalidated
Deallocation





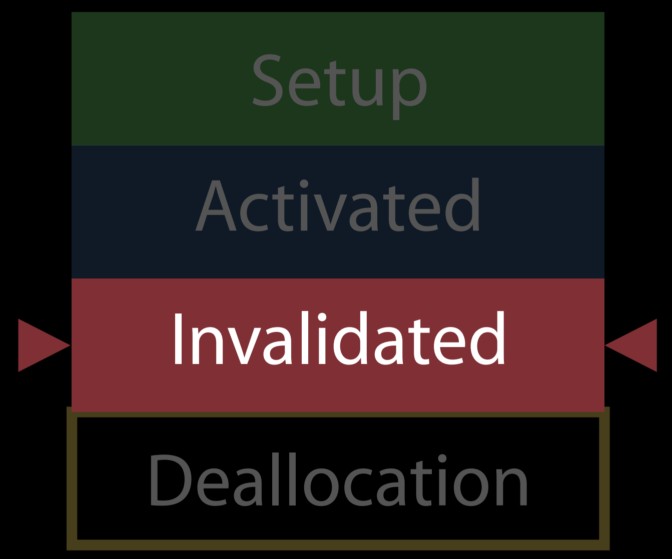
✗ Abandoned memory

✗ Deadlocks

```
// Deadlocks on Serial Queues Assert

Application Specific Information:
BUG IN CLIENT OF LIBDISPATCH: dispatch_barrier_sync called on queue already owned by current
thread

Thread 1 Crashed:: Dispatch queue: com.example.queue
0   libdispatch.dylib               0x00007fff920b44ee _dispatch_barrier_sync_f_slow + 675
1   <YOUR APP>                      0x000000010a3d7f26 __main_block_invoke_2 + 38
2   libdispatch.dylib               0x00007fff920a8ed6 _dispatch_client_callout + 8
3   libdispatch.dylib               0x00007fff920a9b0e _dispatch_barrier_sync_f_invoke + 83
4   <YOUR APP>                      0x000000010a3d7ef6 __main_block_invoke + 38
5   libdispatch.dylib               0x00007fff920b1d54 _dispatch_call_block_and_release + 12
6   libdispatch.dylib               0x00007fff920a8ed6 _dispatch_client_callout + 8
7   libdispatch.dylib               0x00007fff920c2d34 _dispatch_queue_serial_drain + 896
...
```

```
// Deadlocks on Serial Queues Assert

Application Specific Information:
BUG IN CLIENT OF LIBDISPATCH: dispatch_barrier_sync called on queue already owned by current
thread

Thread 1 Crashed:: Dispatch queue: com.example.queue
0   libdispatch.dylib              0x00007fff920b44ee _dispatch_barrier_sync_f_slow + 675
1   <YOUR APP>                     0x000000010a3d7f26 __main_block_invoke_2 + 38
2   libdispatch.dylib              0x00007fff920a8ed6 _dispatch_client_callout + 8
3   libdispatch.dylib              0x00007fff920a9b0e _dispatch_barrier_sync_f_invoke + 83
4   <YOUR APP>                     0x000000010a3d7ef6 __main_block_invoke + 38
5   libdispatch.dylib              0x00007fff920b1d54 _dispatch_call_block_and_release + 12
6   libdispatch.dylib              0x00007fff920a8ed6 _dispatch_client_callout + 8
7   libdispatch.dylib              0x00007fff920c2d34 _dispatch_queue_serial_drain + 896
...
```

Explicit Invalidation



```
class BusyController: SubsystemObserving {

    func invalidate() {



    }

    deinit {

    }
}
```

Explicit Invalidation

Setup

Activated

Invalidated

Deallocation

```
class BusyController: SubsystemObserving {

    func invalidate() {


        DataTransform.sharedInstance.unregister(observer: self)
    }

    deinit {

    }
}
```

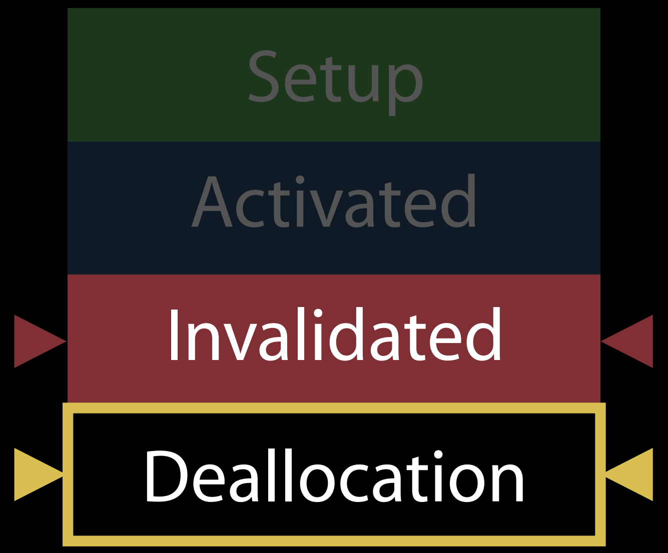
Explicit Invalidation

Setup
Activated
▶ Invalidated ◀
Deallocation

```
class BusyController: SubsystemObserving {

    func invalidate() {
        dispatchPrecondition(.onQueue(DispatchQueue.main))

        DataTransform.sharedInstance.unregister(observer: self)
    }

    deinit {


    }
}
```

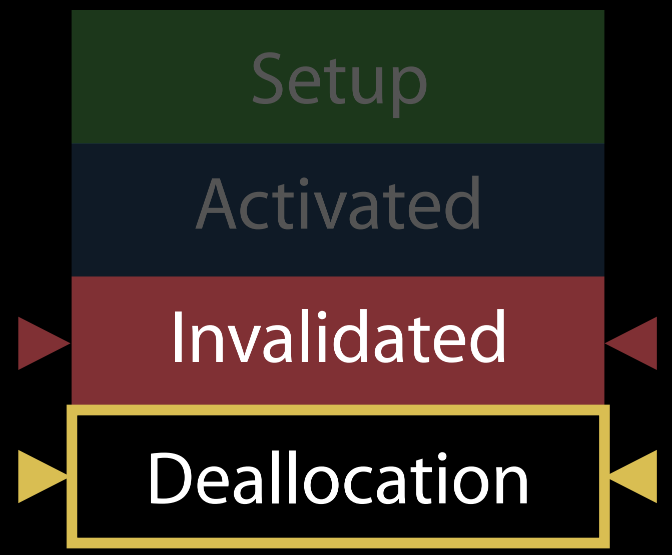
✓

Invalidation as a State



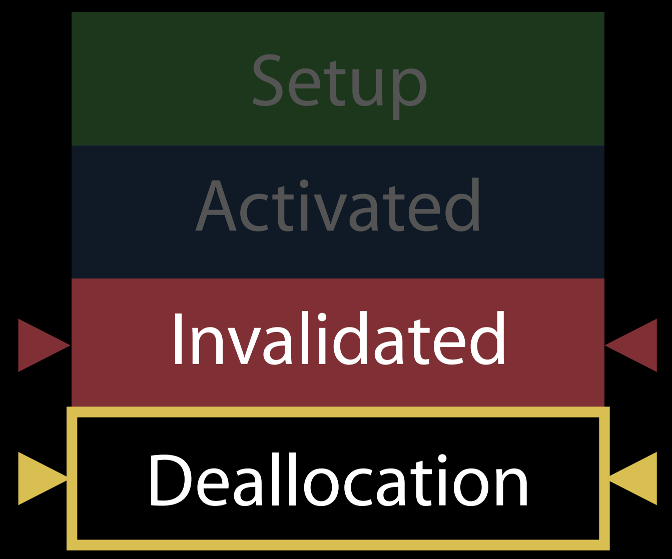
```
class BusyController: SubsystemObserving {

   func invalidate() {
      dispatchPrecondition(.onQueue(DispatchQueue.main))

      DataTransform.sharedInstance.unregister(observer: self)
   }

   deinit {

   }
}
```

Invalidation as a State


Setup
Activated
Invalidated
Deallocation

```
class BusyController: SubsystemObserving {
    private var invalidated: Bool = false
    func invalidate() {
        dispatchPrecondition(.onQueue(DispatchQueue.main))
        invalidated = true
        DataTransform.sharedInstance.unregister(observer: self)
    }

    deinit {
        precondition(invalidated)
    }
}
```

Invalidation as a State



Setup

Activated

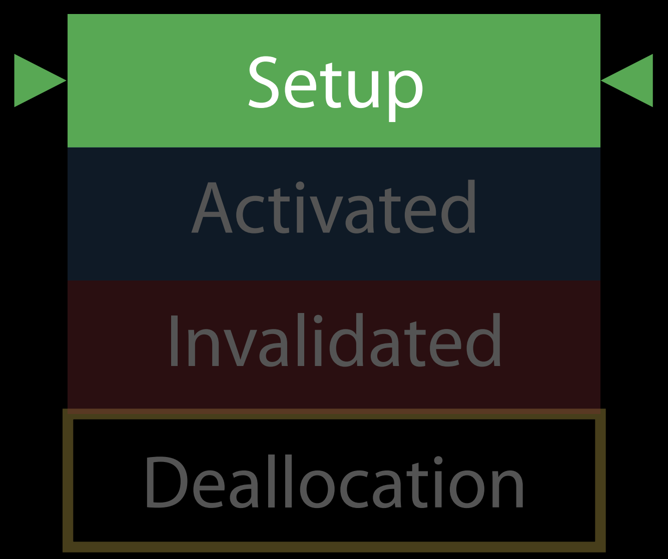
Invalidated

Deallocation

```
class BusyController: SubsystemObserving {
    private var invalidated: Bool = false

    func systemStarted(...) {
        if invalidated { return }
        /* ... */
    }

    deinit {
        precondition(invalidated)
    }
}
```

GCD Object Lifecycle

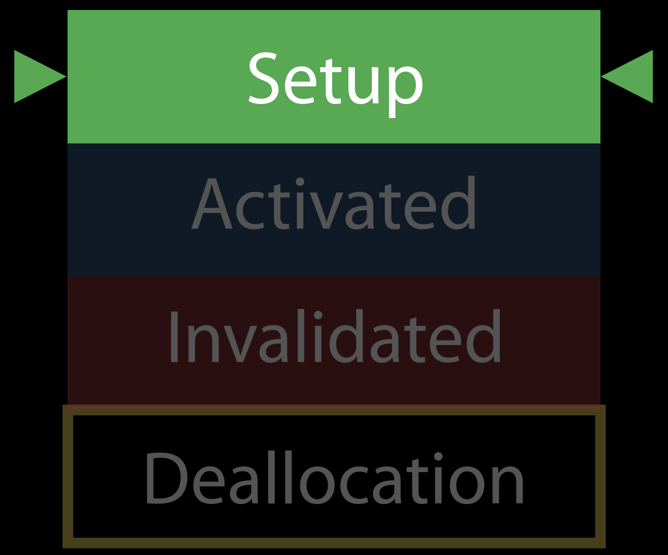
Setup



Attributes and target queue

```
let q = DispatchQueue(label: "com.example.queue", attributes: [.autoreleaseWorkItem])

let source = DispatchSource.read(fileDescriptor: fd, queue: q)
```

Setup

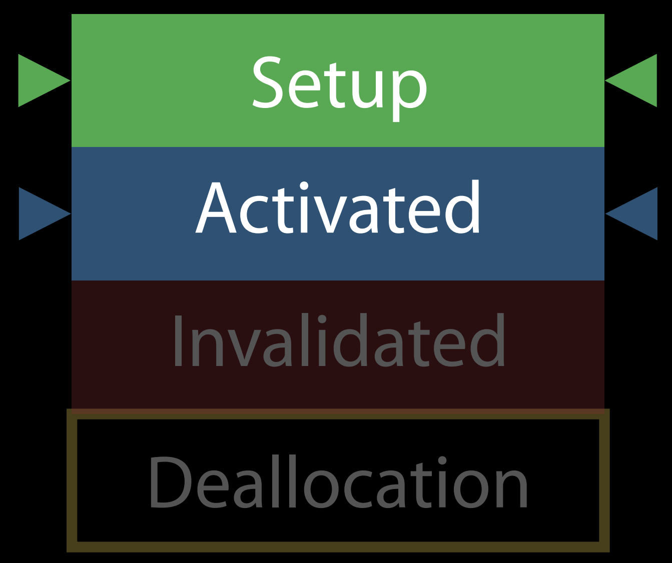

Setup
Activated
Invalidated
Deallocation

Attributes and target queue

Source handlers

```
let q = DispatchQueue(label: "com.example.queue", attributes: [.autoreleaseWorkItem])

let source = DispatchSource.read(fileDescriptor: fd, queue: q)

source.setEventHandler { /* handle your event here */ }
source.setCancelHandler { close(fd) }
```

Activation

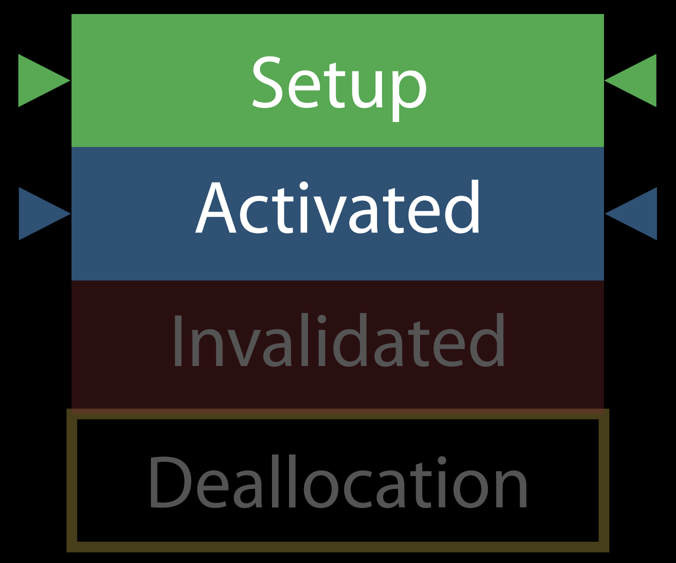


Properties of dispatch objects must not be mutated after activation

```
extension DispatchObject {
   func activate()
}
```

Activation

NEW

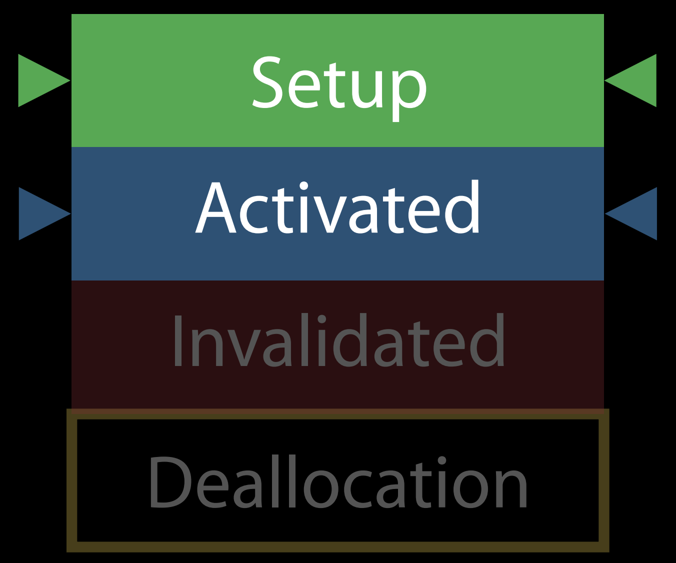


Properties of dispatch objects must not be mutated after activation

```
extension DispatchObject {
   func activate()
}
```

Activation

NEW



Properties of dispatch objects must not be mutated after activation

- Queues can also be created inactive

```swift
extension DispatchObject {
    func activate()
}

let queue = DispatchQueue(label: “com.example.queue”, attributes: [.initiallyInactive])
```

Cancellation

Setup

Activated

Invalidated

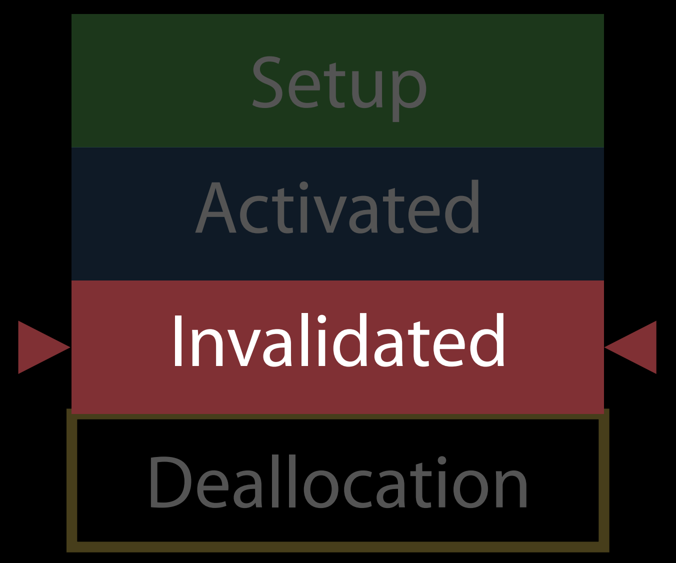
Deallocation

Sources require explicit cancellation

- Event monitoring is stopped

```
extension DispatchSource {
   func cancel()
}
```

Cancellation

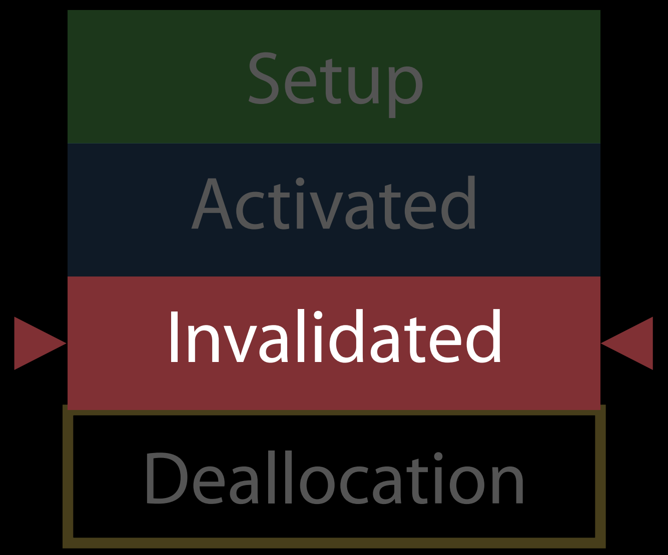

Setup
Activated
Invalidated
Deallocation

Sources require explicit cancellation

- Event monitoring is stopped
- Cancellation handler runs

```
let source = DispatchSource.read(fileDescriptor: fd, queue: q)

source.setCancelHandler { close(fd) }
```

Cancellation



Setup

Activated

Invalidated

Deallocation

Sources require explicit cancellation

- Event monitoring is stopped
- Cancellation handler runs
- All handlers are deallocated

```
let source = DispatchSource.read(fileDescriptor: fd, queue: q)

source.setCancelHandler { close(fd) }
```

Deallocation Hygiene

Setup
Activated
Invalidated
▶ Deallocation ◀

GCD Objects expect to be in a defined state at deallocation

- Activated
- Not suspended

Summary

Organize your application around data flows into independent subsystems

Synchronize state with Dispatch Queues

Use the activate/invalidate pattern

More Information

https://developer.apple.com/wwdc16/720

Related Sessions

Thread Sanitizer and Static Analysis	Mission	Thursday 10:00AM
Going Server-side with Swift Open Source	Mission	Friday 9:00AM
Optimizing I/O for Performance and Battery Life	Nob Hill	Friday 11:00AM

Labs

GCD Lab	Frameworks Lab D	Tuesday 5:00PM

WWDC16