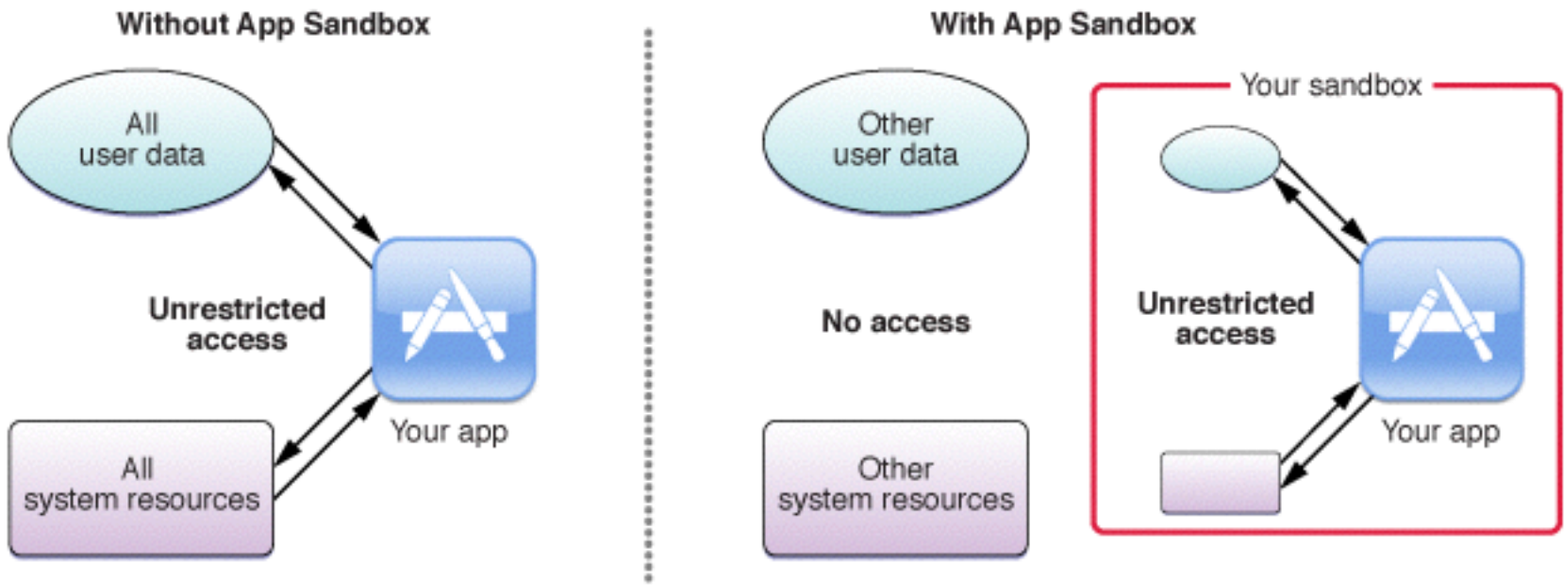
Extensions: Sharing Code and Data

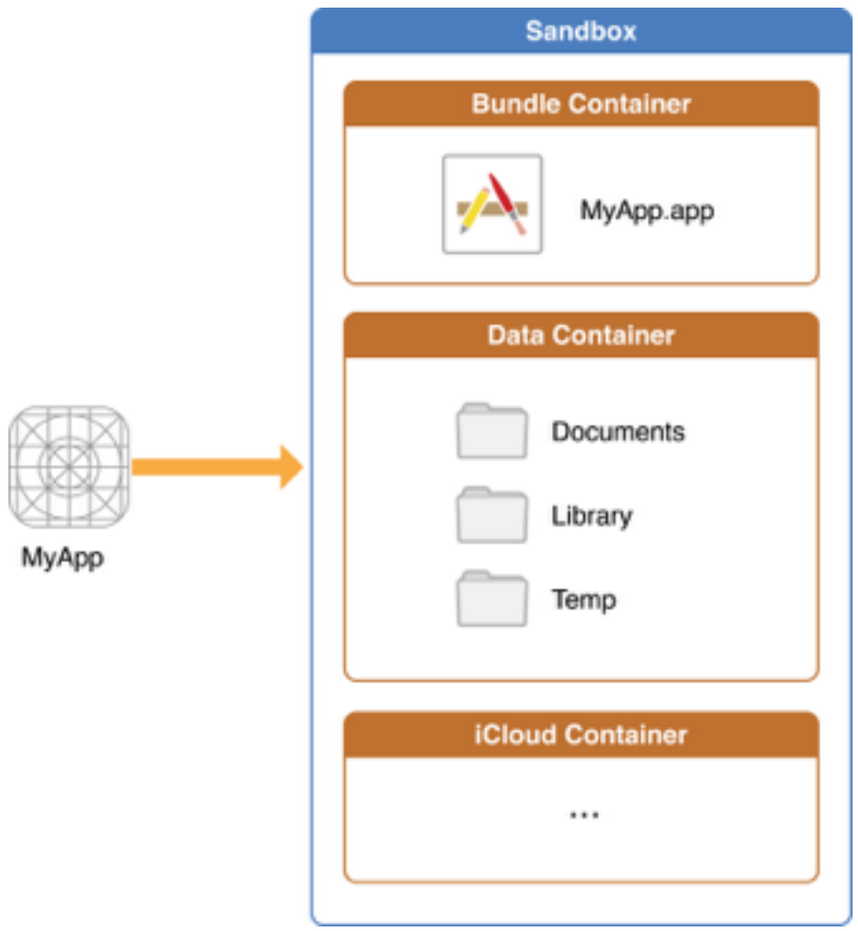
CS193W - Spring 2016 - Lecture 6

Sandboxing



- Malicious apps cannot access resources outside their sandbox.

Sandboxed Content



Read-only

Read/write

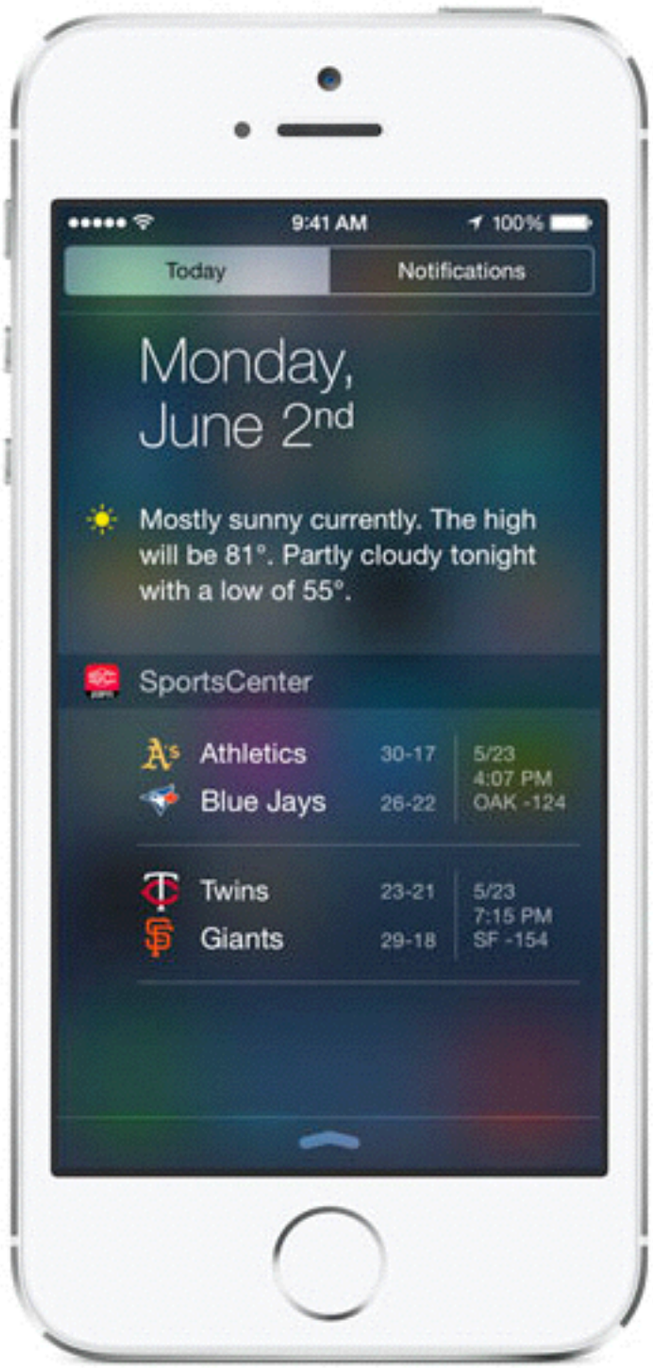
Read/write

App Extensions

- App Extensions (e.g. Apple Watch applications) allow an app to extend the functionality of other apps

- They do this while keeping apps sandboxed. Communication occurs via an Extension Context.

Today Widgets

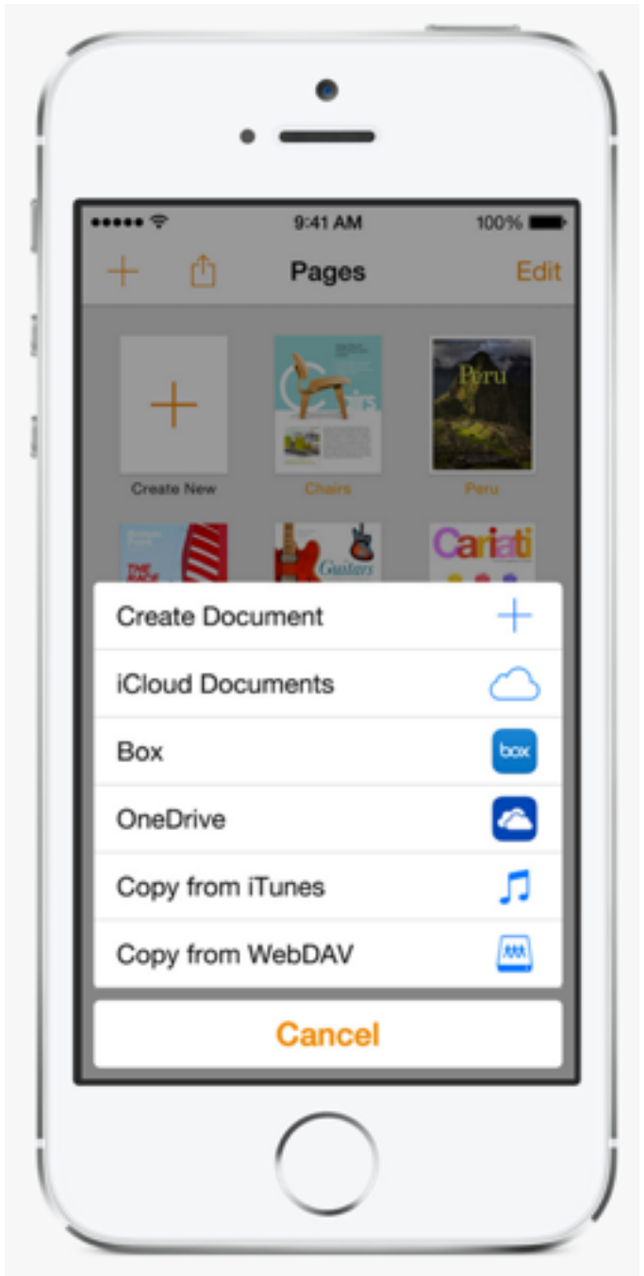


Share Extensions

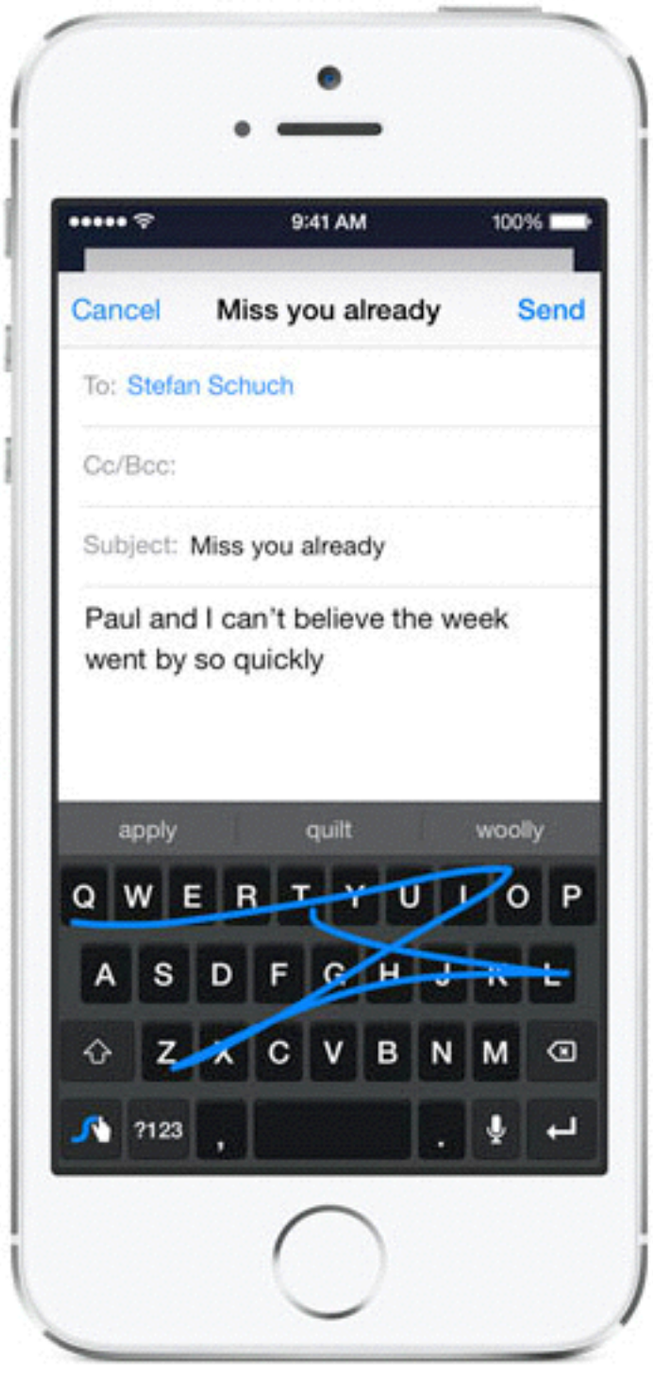
Action Extensions

Photo Editors

Document Providers



Custom Keyboards



Extensions are not Apps

- Do not have application lifecycle events.

- They do not have application delegates or application lifecycle events.

Some APIs are unavailable to App Extensions

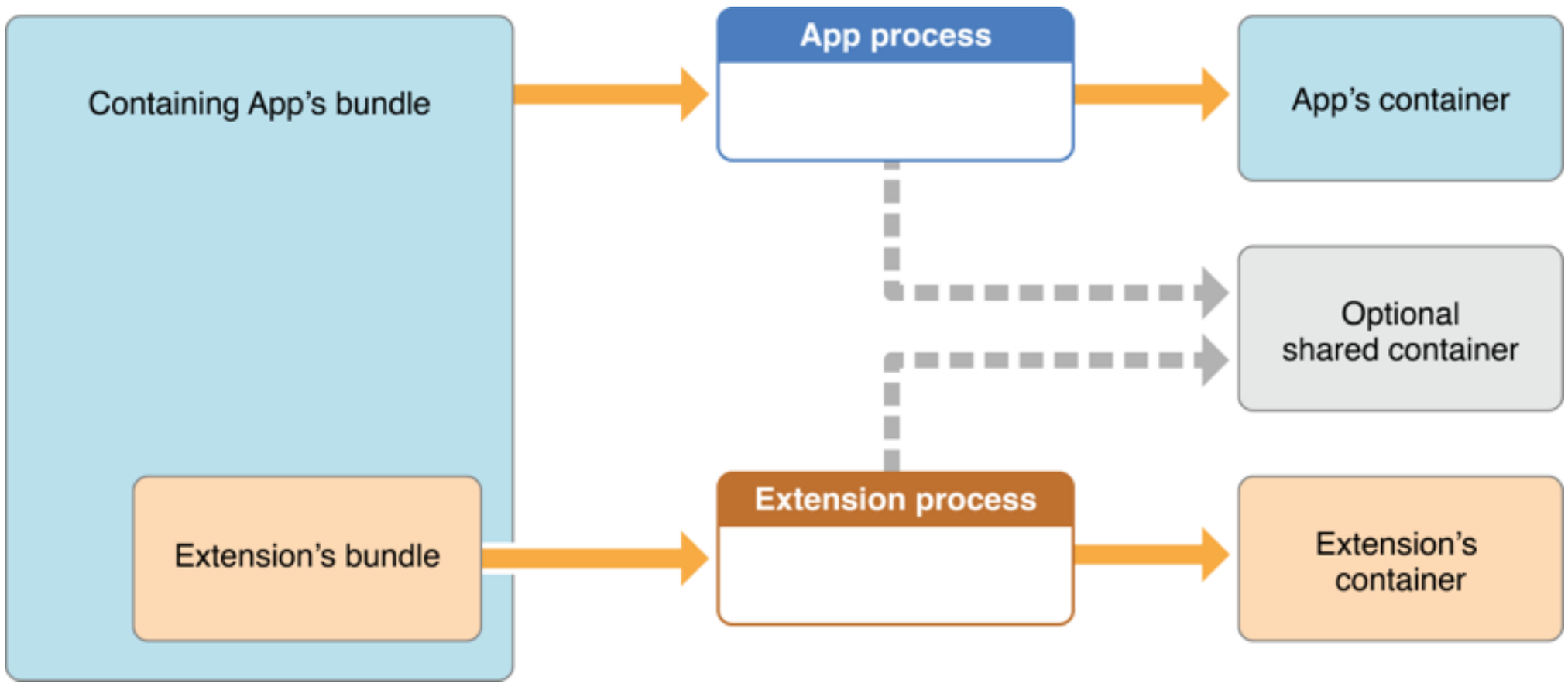
- There is no sharedApplication object (i.e. no App Delegate)

- Cannot access the camera or microphone of the iOS device

- Cannot perform long-running background tasks

- Cannot access APIs marked NS_EXTENSION_UNAVAILABLE (HealthKit, EventKit, etc.)

Application Structure



Sharing Data (Extensions that run on the iPhone)

- You can share data via NSUserDefaults
- Instead of NSUserDefaults.standardUserDefaults(), use:

NSUserDefaults(suiteName: "group.com.mycompany.myapp"]

aka

[NSUserDefauts initWithSuiteName: @“group.com.mycompany.myapp"]

where group.com.mycompany.myapp is the id of an App Group

Setting up App Groups

- Go to developer.apple.com -> Certificates, Identifiers & Profiles

- Create App IDs for the containing app and the extensions and enable App Groups services for them:
    - com.mycompany.myapp
    - com.mycompany.myapp.myextension
- Then create an app group:
    - group.com.mycompany.myapp

Apple Inc.

Developer

Technologies Resources Programs Support Member Center Search Developer

Certificates, Identifiers & Profiles

Michael Kassoff

iOS Apps

- **Certificates**
  - All
  - Pending
  - Development
  - Production
- **Identifiers**
  - App IDs
  - Pass Type IDs
  - Website Push IDs
  - iCloud Containers
  - App Groups
  - Merchant IDs
- **Devices**
  - All
- **Provisioning Profiles**
  - All
  - Development
  - Distribution

Register iOS App ID

ID

Registering an App ID

The App ID string contains two parts separated by a period (.)—an App ID Prefix that is defined as your Team ID by default and an App ID Suffix that is defined as a Bundle ID search string. Each part of an App ID has different and important uses for your app. Learn More

App ID Description

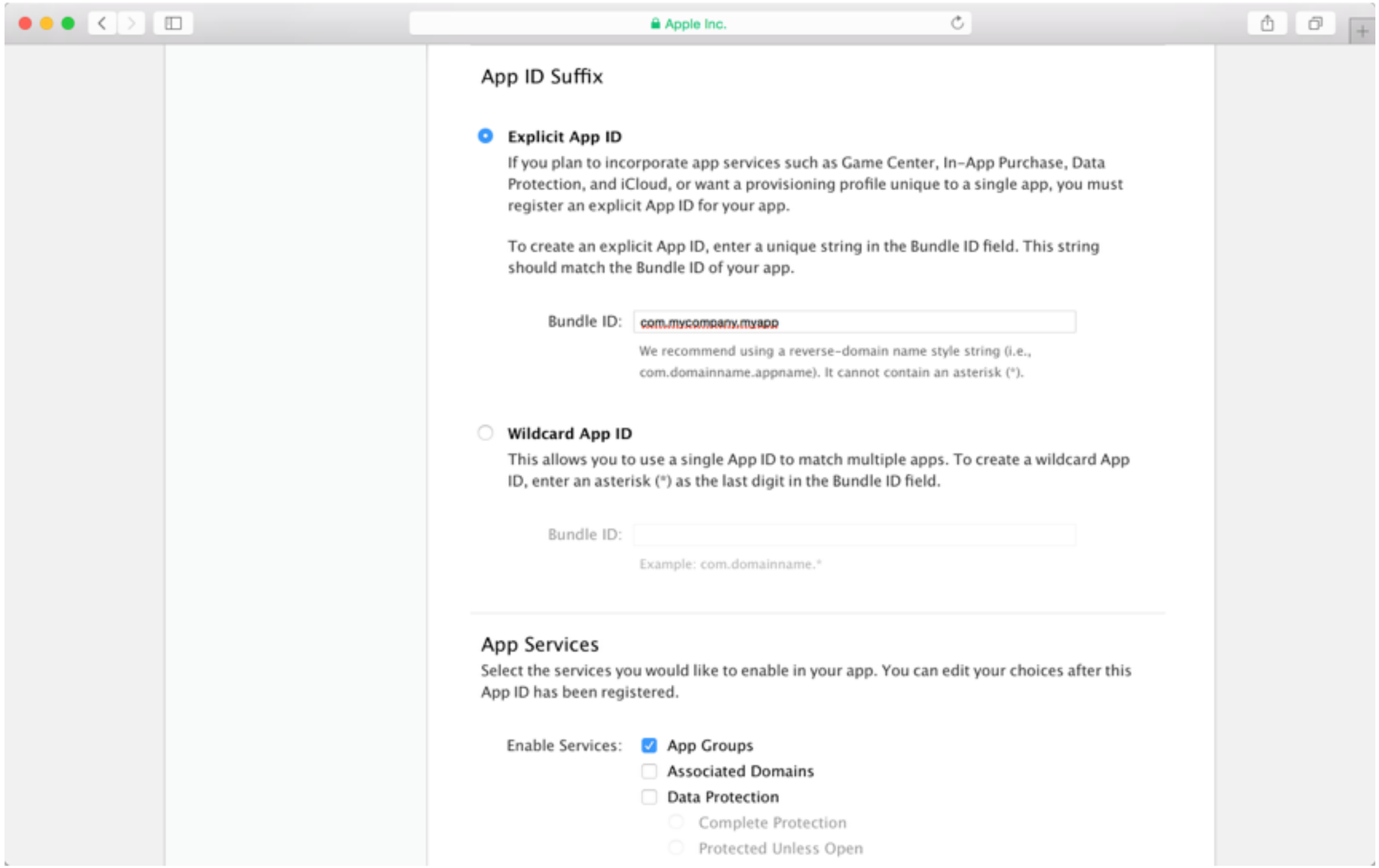
Name: My App ID

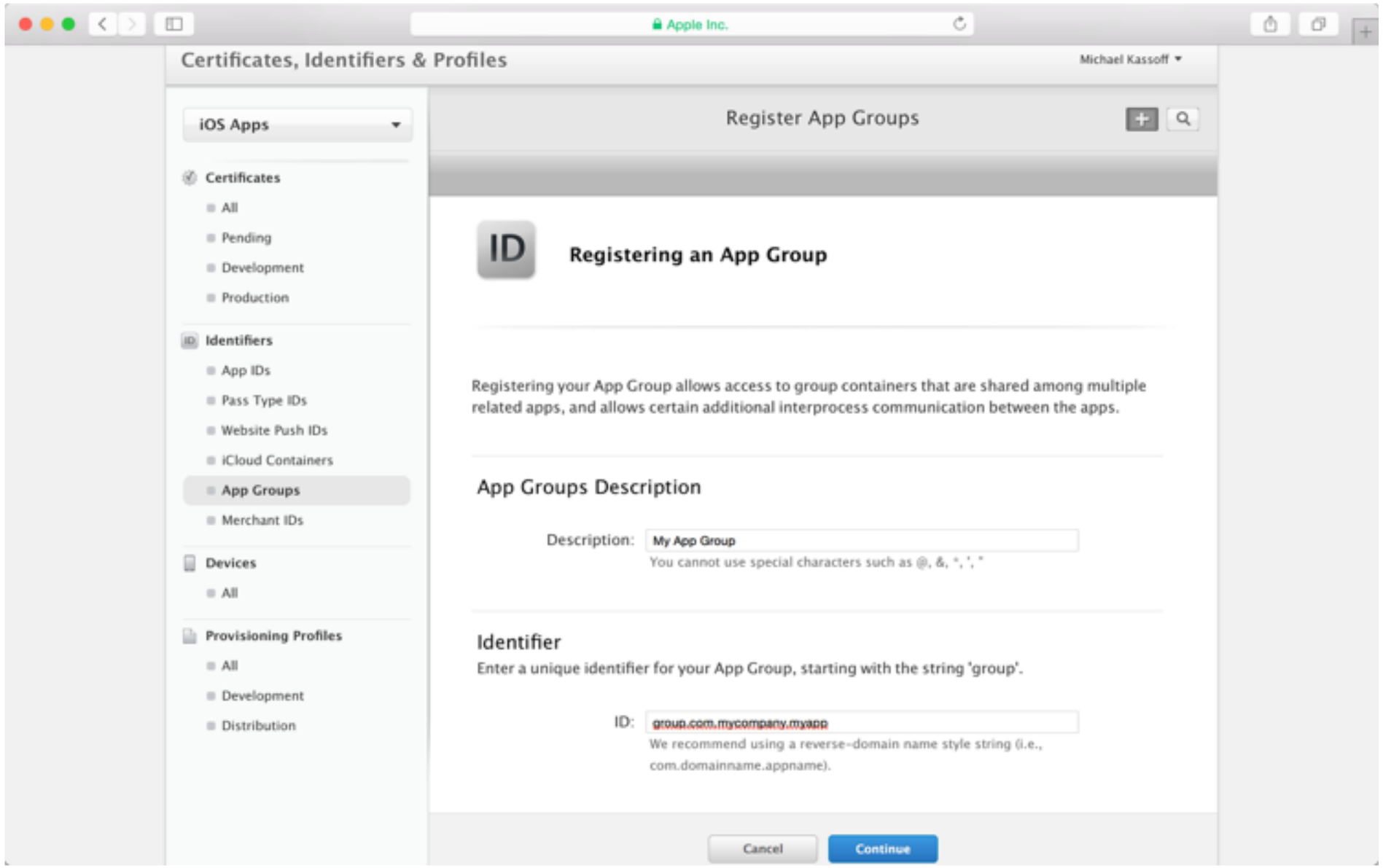
You cannot use special characters such as @, &, *, ', "

App ID Prefix

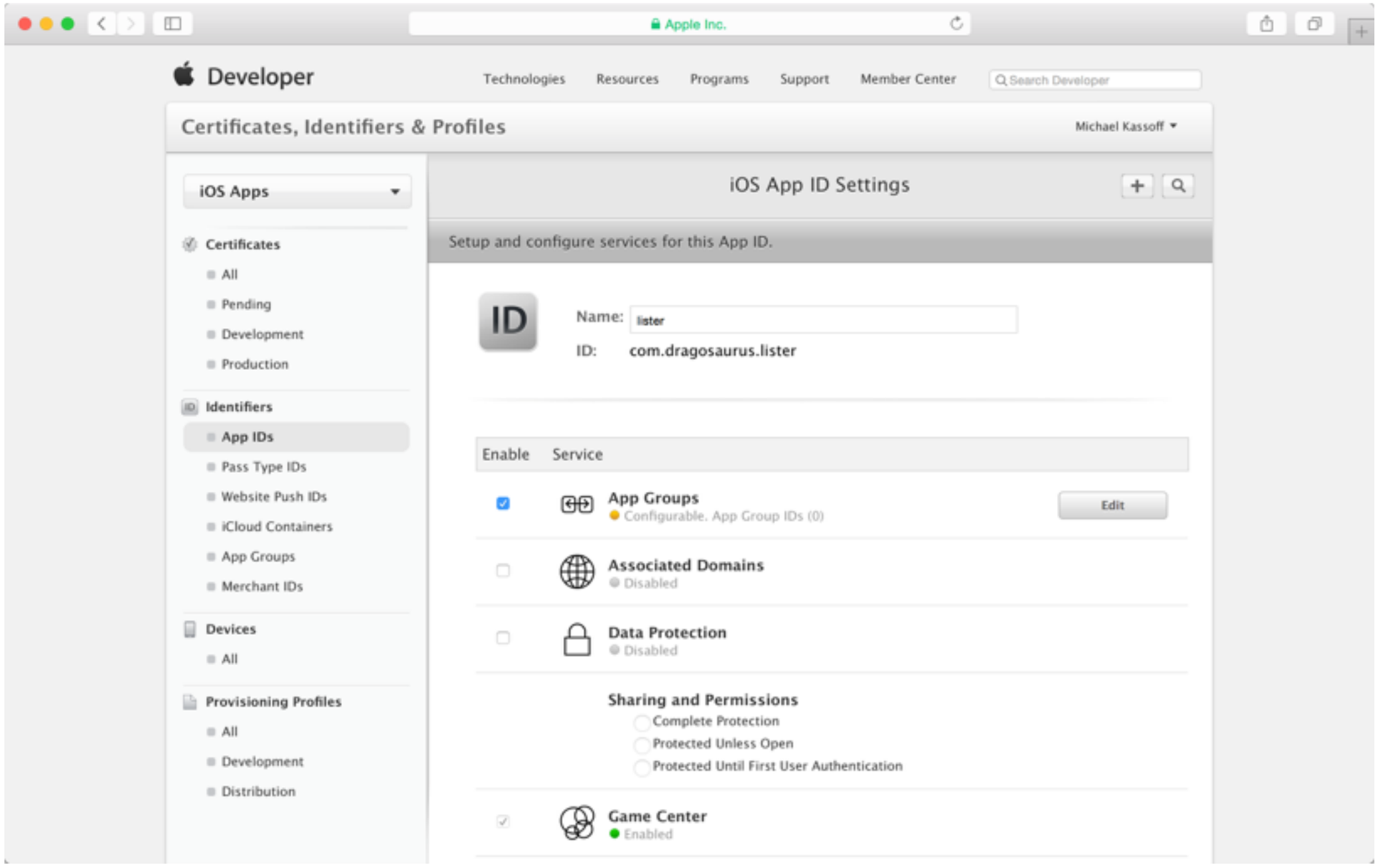
Value: VS88UM9Z3U (Team ID)

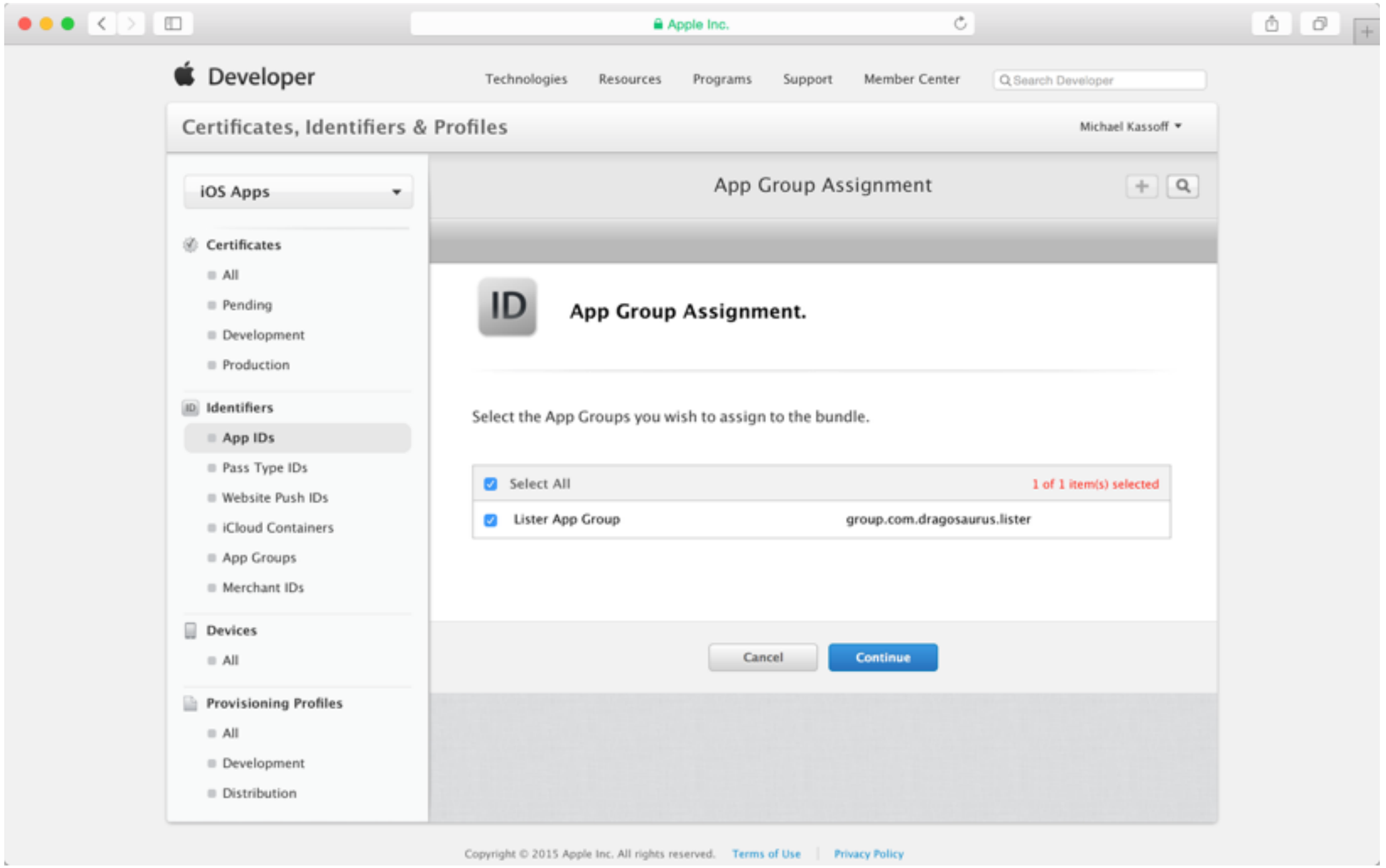
App ID Suffix


Apple Inc.
App ID Suffix
Explicit App ID
If you plan to incorporate app services such as Game Center, In-App Purchase, Data Protection, and iCloud, or want a provisioning profile unique to a single app, you must register an explicit App ID for your app.
To create an explicit App ID, enter a unique string in the Bundle ID field. This string should match the Bundle ID of your app.
Bundle ID:
com.mycompany.myapp
We recommend using a reverse-domain name style string (i.e., com.domainname.appname). It cannot contain an asterisk (*).
Wildcard App ID
This allows you to use a single App ID to match multiple apps. To create a wildcard App ID, enter an asterisk (*) as the last digit in the Bundle ID field.
Bundle ID:
Example: com.domainname.*
App Services
Select the services you would like to enable in your app. You can edit your choices after this App ID has been registered.
Enable Services:
App Groups
Associated Domains
Data Protection
Complete Protection
Protected Unless Open


Apple Inc.
Certificates, Identifiers & Profiles
Michael Kassoff
iOS Apps
Certificates
All
Pending
Development
Production
Identifiers
App IDs
Pass Type IDs
Website Push IDs
iCloud Containers
App Groups
Merchant IDs
Devices
All
Provisioning Profiles
All
Development
Distribution
Register App Groups
ID
Registering an App Group
Registering your App Group allows access to group containers that are shared among multiple related apps, and allows certain additional interprocess communication between the apps.
App Groups Description
Description:
My App Group
You cannot use special characters such as @, &, *, ', "
Identifier
Enter a unique identifier for your App Group, starting with the string 'group'.
ID:
group.com.mycompany.myapp
We recommend using a reverse-domain name style string (i.e., com.domainname.appname).
Cancel
Continue

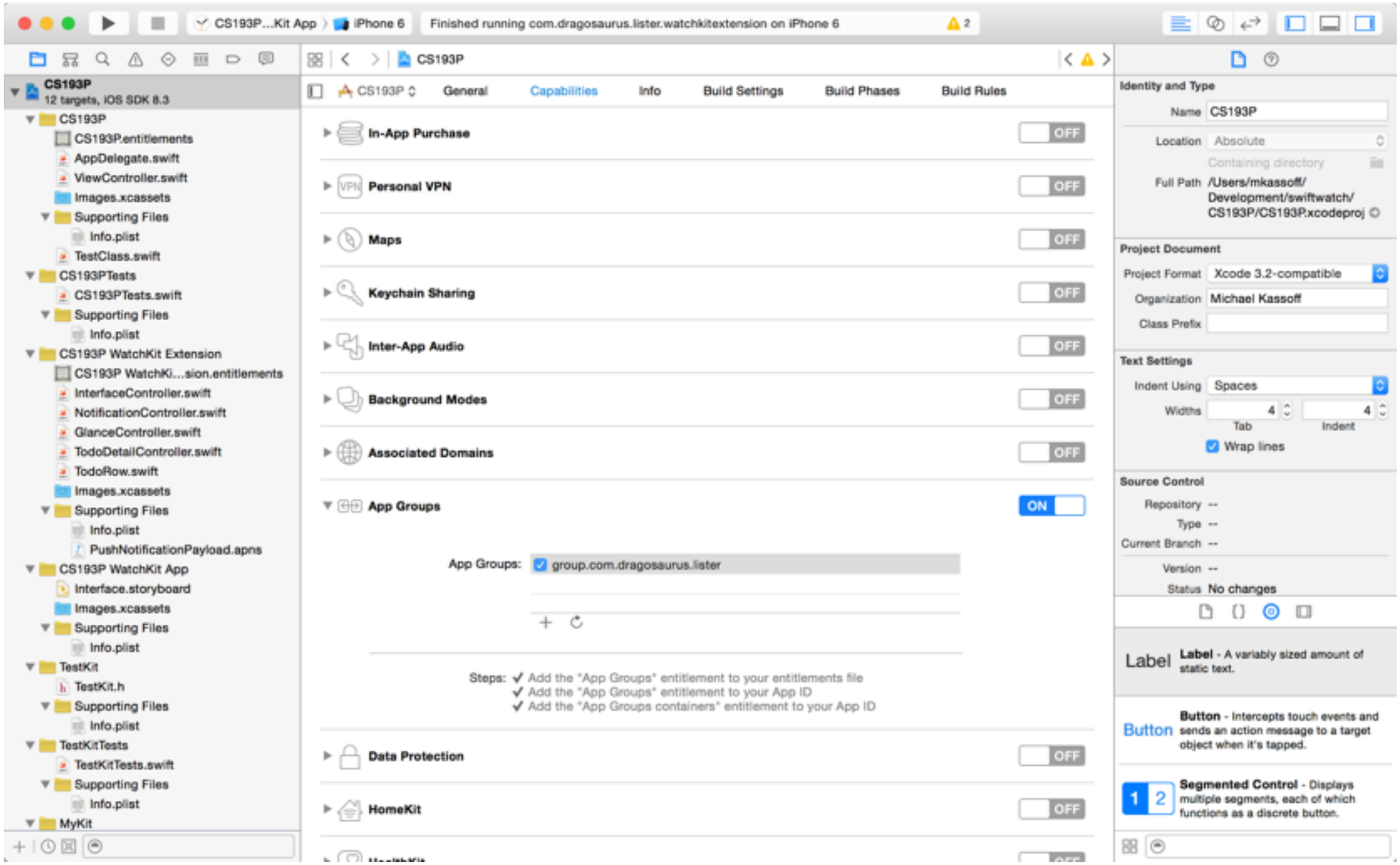
Associating App IDs with App Groups




Apple Inc.
Developer
Technologies
Resources
Programs
Support
Member Center
Search Developer
Certificates, Identifiers & Profiles
Michael Kassoff
iOS Apps
Certificates
All
Pending
Development
Production
Identifiers
App IDs
Pass Type IDs
Website Push IDs
iCloud Containers
App Groups
Merchant IDs
Devices
All
Provisioning Profiles
All
Development
Distribution
App Group Assignment
ID
App Group Assignment.
Select the App Groups you wish to assign to the bundle.
Select All
1 of 1 item(s) selected
Lister App Group
group.com.dragosaurus.lister
Cancel
Continue
Copyright © 2015 Apple Inc. All rights reserved.
Terms of Use
Privacy Policy

Setting up App Groups (cont'd)

- In Xcode, enable App Groups for each target for that will be sharing data between via the app group


CS193P...Kit App
iPhone 6
Finished running com.dragosaurus.lister.watchkitextension on iPhone 6
CS193P
CS193P
12 targets, iOS SDK 8.3
CS193P
CS193P.entitlements
AppDelegate.swift
ViewController.swift
Images.xcassets
Supporting Files
Info.plist
TestClass.swift
CS193PTests
CS193PTests.swift
Supporting Files
Info.plist
CS193P WatchKit Extension
CS193P WatchKi....sion.entitlements
InterfaceController.swift
NotificationController.swift
GlanceController.swift
TodoDetailController.swift
TodoRow.swift
Images.xcassets
Supporting Files
Info.plist
PushNotificationPayload.apns
CS193P WatchKit App
Interface.storyboard
Images.xcassets
Supporting Files
Info.plist
TestKit
TestKit.h
Supporting Files
Info.plist
TestKitTests
TestKitTests.swift
Supporting Files
Info.plist
MyKit
CS193P
CS193P
General
Capabilities
Info
Build Settings
Build Phases
Build Rules
In-App Purchase
OFF
Personal VPN
OFF
Maps
OFF
Keychain Sharing
OFF
Inter-App Audio
OFF
Background Modes
OFF
Associated Domains
OFF
App Groups
ON
App Groups:
group.com.dragosaurus.lister
Steps:
Add the "App Groups" entitlement to your entitlements file
Add the "App Groups" entitlement to your App ID
Add the "App Groups containers" entitlement to your App ID
Data Protection
OFF
HomeKit
OFF
Identity and Type
Name
CS193P
Location
Absolute
Containing directory
Full Path
/Users/mkassoff/Development/swiftwatch/CS193P/CS193P.xcodeproj
Project Document
Project Format
Xcode 3.2-compatible
Organization
Michael Kassoff
Class Prefix
Text Settings
Indent Using
Spaces
Widths
4
4
Tab
Indent
Wrap lines
Source Control
Repository --
Type --
Current Branch --
Version --
Status No changes
Label
Label - A variably sized amount of static text.
Button
Button - Intercepts touch events and sends an action message to a target object when it's tapped.
1 2
Segmented Control - Displays multiple segments, each of which functions as a discrete button.

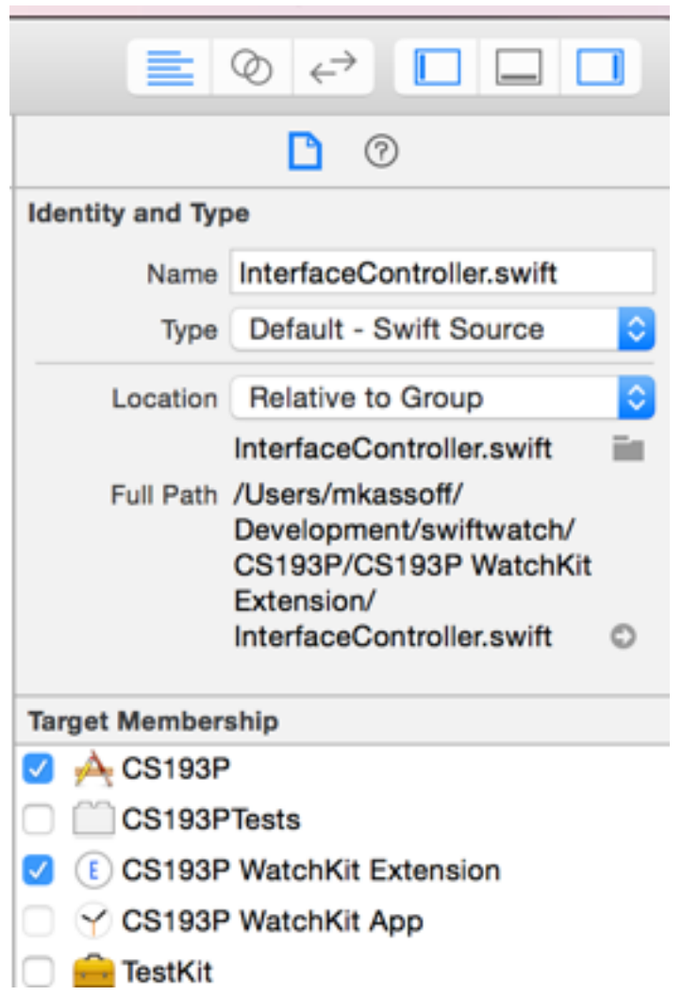
Sharing Code

- To share code and resources between targets, you can either:

  Include the source file or resource in each target you want it in

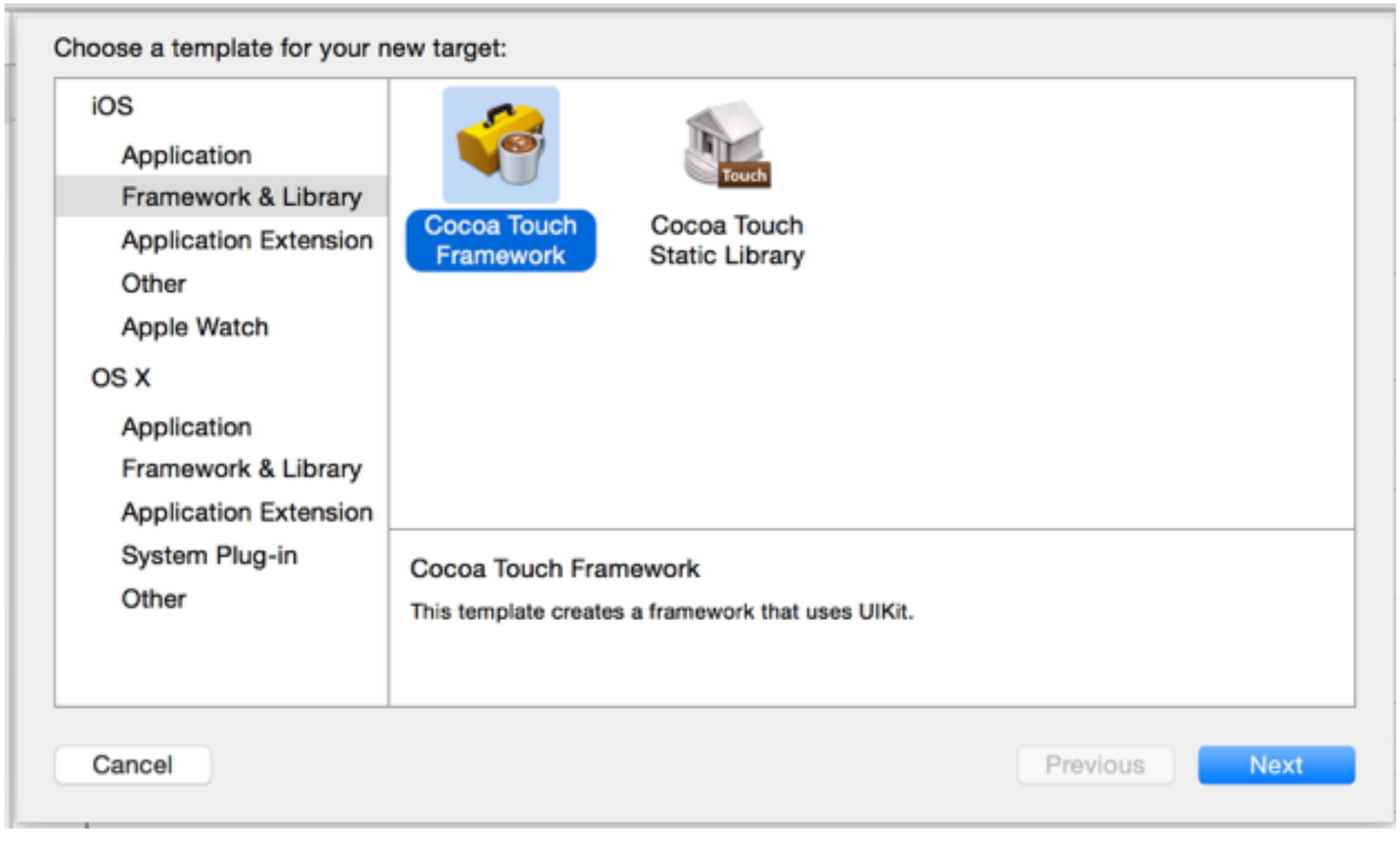
  -OR-

  Create an embedded framework that contains a single copy of your code or resource



Creating a new framework

- File -> New Target -> Framework & Library -> Cocoa Touch Framework



Choose options for your new target:

Product Name: SampleKit

Organization Name: Michael Kassoff

Organization Identifier: com.dragosaurus.lister

Bundle Identifier: com.dragosaurus.lister.SampleKit

Language: Swift

Project: CS193P

Embed in Application: CS193P

Cancel

Previous

Finish

Using your Framework

1. Add classes to your framework

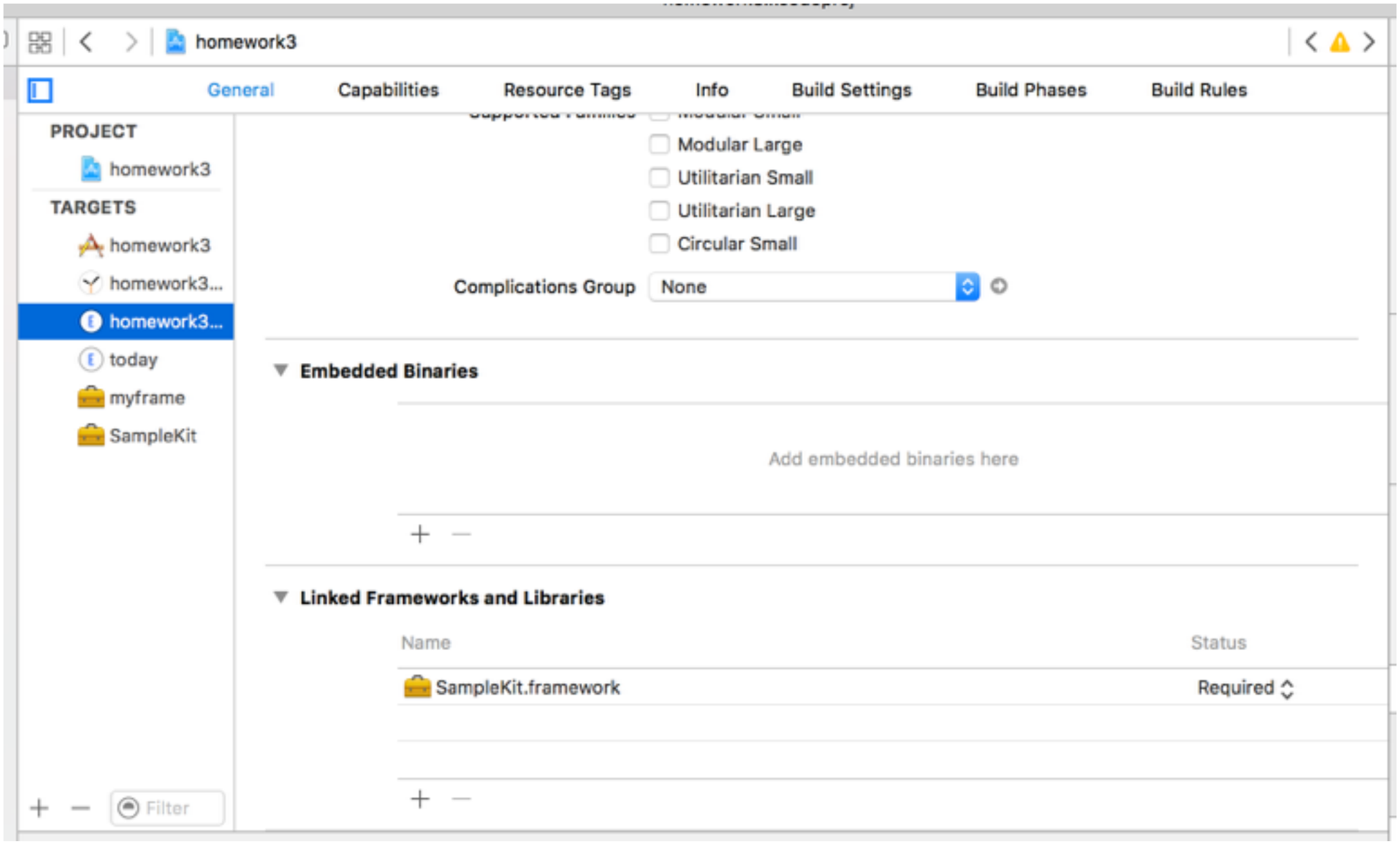
2. Mark classes, methods and enumerations that you want to expose as public

3. Add framework as Linked Framework to your target (see next slide)

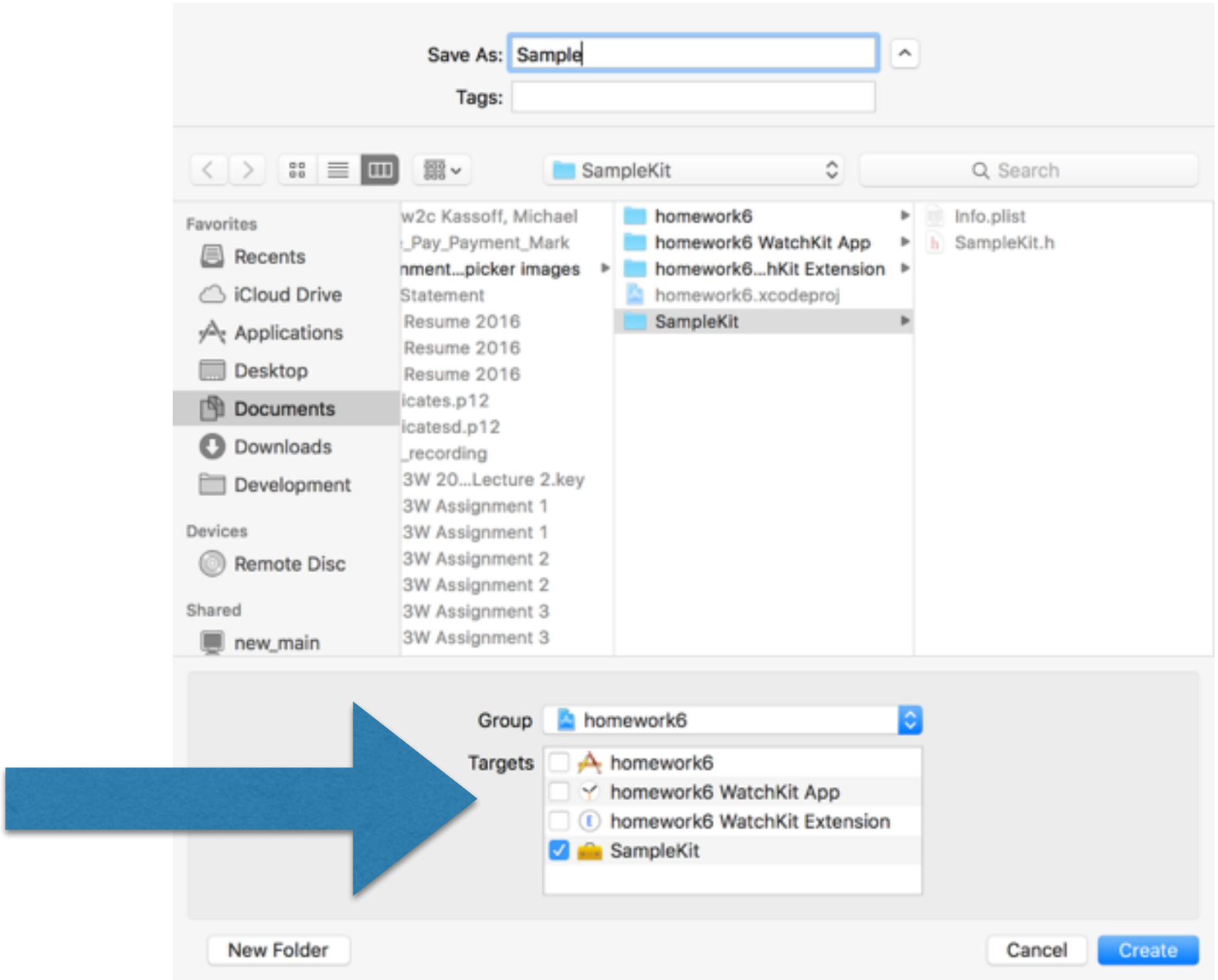
4. Import your framework with an import statement:

```
import SampleKit
```

Adding Framework to a Target



Adding a File to Your Framework



Access Control in Swift: Terminology

- *Module* - A framework or application. Each build target corresponds to a module.

- *Source File* - A single file of code in a module. Note that a single file can contain multiple class definitions etc.

Access Levels

- **Public** - An entity that can be accessed from outside and inside a module

- **Internal** - An entity that can be accessed only inside a module

- **Private** - An entity that can be accessed only within its own source file

Access control: Syntax

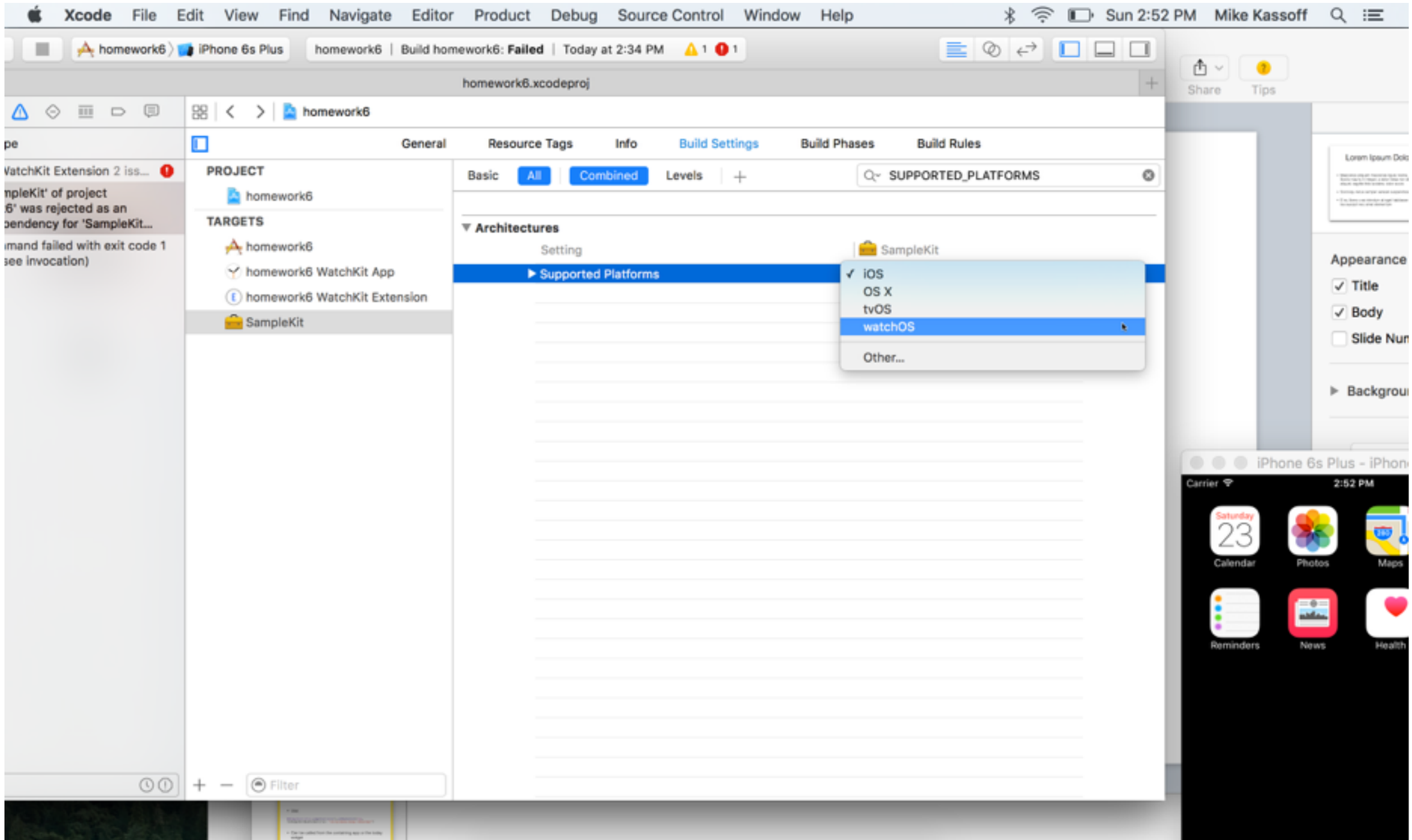
```
public class SomePublicClass {}
internal class SomeInternalClass {}
private class SomePrivateClass {}

public var somePublicVariable = 0
internal let someInternalConstant = 0
private func somePrivateFunction() {}
```

Example

```
public class Sample {
    public func doSomething() {
        print("something")
    }
}
```

But: Frameworks are Tied to an OS



So…

- You can use Frameworks to share code between a Today Extension and an iOS app

- But you can’t use Frameworks to share code between an iOS app and a Watch App

- You can still use Frameworks to share code across apps

Watch Connectivity

Watch App, iPhone App

- Your iPhone app and your Apple Watch app often need to share data and talk to one another

- In particular, this is important for apps or data that are not saved in the cloud

WatchConnectivity Framework

- Both iOS and WatchOS share the WatchConnectivity framework

- Requires iOS 9 and WatchOS 2

- The majority of methods in the framework are found on both iOS and WatchOS

- Some functionality has changed / been added in iOS 9.3 / WatchOS 2.2 to allow multiple watches to be paired with the same iPhone

2 Types of Transfers

Interactive messaging

- For when you need the transfer to happen right away
- Requires both the phone and watch to be reachable

Background transfers

- For when you can afford to wait
- WatchConnectivity batches transfers to save battery life

WCSession

- Most of the functionality of the WatchConnectivity framework is found in the **`WCSession`** class

- Both the iOS app and WatchOS app must maintain their own **`WCSession`** object

Getting the Session

- Call `WCSession.isSupported()` on iPhone to make sure that the iPhone is a model that can pair with an Apple Watch. This method will always return `true` on an Apple Watch.

- If so, you can access the session by calling `WCSession.defaultSession()`

Activating the Session

- Before activating your session you must assign a `delegate` that conforms to `WCSessionDelegate`

- Then you can call `session.activateSession()`

- After calling `activateSession`, your delegate will get a callback on:

```
optional func session(_ session: WCSession,
activationDidCompleteWithState activationState:
WCSessionActivationState, error error: NSError?)
```

- `activationState` can be one of `.NotActivated`, `.Inactive` and `.Activated`

Paired Device Information (iOS only)

Once you have an activated session, you can call:

`paired` - `true` if the iPhone is currently paired with an Apple Watch

`watchAppInstalled` - `true` if the app is installed on the Apple Watch

`complicationEnabled` – `true` if the user has the watch app's complication enabled

`true` if the app's complication is installed on the active watch face

Reachability

`reachable`

`true` when other device is paired, in-range and has an active session. iOS apps do not need to be in the foreground but WatchOS apps do.

`iOSDeviceNeedsUnlockAfterRebootForReachability`

`true` if the user's iPhone has not been unlocked yet since rebooting. (WatchOS only)

Interactive Messaging

- The devices must be reachable
- One side sends with WCSession

```
sendMessage(_:replyHandler:errorHandler:)
```

- The other side receives with WCSessionDelegate

```
session(_:didReceiveMessage:)
```

Sending a Message

```
func sendMessage(_ message: [String : AnyObject],
    replyHandler replyHandler: (([String : AnyObject]) -> Void)?,
    errorHandler errorHandler: ((NSError) -> Void)?)
```

`message` and the reply are dictionaries of property list values.

Set the `replyHandler` to `nil` if you don't want a reply.

The error handler is invoked if the device you are sending to is unreachable.

Messages are queued in order sent and sent asynchronously. Reply callbacks occur serially on a background thread.

Sending to iOS wakes up the iOS app in the background, but sending to WatchOS requires that the WatchOS app is in the foreground already.

Receiving a Message

If no reply is requested

```
optional func session(_ session: WCSession,
    didReceiveMessage message: [String : AnyObject])
```

If a reply is requested

```
optional func session(_ session: WCSession,
    didReceiveMessage message: [String : AnyObject],
        replyHandler replyHandler: ([String :
AnyObject]) -> Void)
```

In this case, you must call `replyHandler` at some point.

Messages are received serially on an background thread.

Sending and Receiving Data

Same idea, but send **`NSData`** instead of a **`Dictionary`**.

```
func sendMessageData(_ data: NSData,
        replyHandler replyHandler: ((NSData) -> Void)?,
        errorHandler errorHandler: ((NSError) -> Void)?)
```

```
optional func session(_ session: WCSession,
didReceiveMessageData messageData: NSData)

optional func session(_ session: WCSession,
didReceiveMessageData messageData: NSData,
        replyHandler replyHandler: (NSData) -> Void)
```

Types of Background Transfers

Updating Application Context
Only the latest context is received by the receiver; previous contexts are overridden. Good for getting a head start on updating your interface with frequently updated data.

e.g. Show 5 most current emails or news stories

Transferring Property Lists
Messages are queued in the order received.

e.g. change a setting

Transferring Files
Messages are queued in the order received.

e.g. transfer a voice message

Updating Application Context

Sender (WCSession)

```
do {
      try session.updateApplicationContext(applicationContext)
    } catch let error {
      throw error
    }
}
```

Receiver (WCSessionDelegate)

```
optional func session(_ session: WCSession,
didReceiveApplicationContext applicationContext: [String : AnyObject])
```

Application Context Properties (WCSession)

`applicationContext`
The latest application context sent

`receivedApplicationContext`
The latest application context received

Sending Property List Data

WCSession

```
func transferUserInfo(_ userInfo: [String : AnyObject]) -> WCSessionUserInfoTransfer
var outstandingUserInfoTransfers: [WCSessionUserInfoTransfer] { get }
```

WCSessionDelegate

```
optional func session(_ session: WCSession,
didFinishUserInfoTransfer userInfoTransfer: WCSessionUserInfoTransfer,
                    error error: NSError?)
```

Transfers happen in the background and continue even if the app is suspended.

Receiving Property List Data

WCSessionDelegate

```
optional func session(_ session: WCSession,
    didReceiveUserInfo userInfo: [String : AnyObject])
```

Monitoring User Info Transfers with WCSessionUserInfoTransfer

`userInfo`
The dictionary being sent

`transferring`
`true` if the data has yet to be transferred completely.
`false` if the transfer is complete.

`cancel()`
cancels the transfer

Sending Complication Data

```
func transferCurrentComplicationUserInfo(_ userInfo: [String : AnyObject]) ->
WCSessionUserInfoTransfer
```

works like `transferUserInfo` except:

- Only used for transferring from iPhone to Apple Watch
- High priority, is sent right away
- Only one of these can be sent at a time; initiating a new one cancels the old one.

Sending Files

WCSession

```
func transferFile(_ file: NSURL,
         metadata metadata: [String : AnyObject]?) -> WCSessionFileTransfer

var outstandingFileTransfers: [WCSessionFileTransfer] { get }
```

WCSessionDelegate

```
optional func session(_ session: WCSession,
didFinishFileTransfer fileTransfer: WCSessionFileTransfer,
                error error: NSError?)
```

File URLs must be local to the sending device.

Transfers happen in the background and continue even if the app is suspended.

Receiving Files

WCSessionDelegate

```
optional func session(_ session: WCSession,
        didReceiveFile file: WCSessionFile)
```

WCSessionFile

`fileURL: NSURL`
The temporary URL of the file has been transferred. You must transfer the file to a permanent directory before `session:didReceiveFile:` returns.

`metadata: [String : AnyObject]?`
The optional dictionary of property list values sent with the file

Monitoring User Info Transfers with WCSessionFileTransfer

`file`
The `WCSessionFile` being sent

`transferring`
`true` if the data has yet to be transferred completely.
`false` if the transfer is complete.

`cancel()`
cancels the transfer