Sample Code

Building an App to Notify Users of COVID-19 Exposure

Inform people when they may have been exposed to COVID-19.

Download

iOS 12.5+ | iPadOS 12.5+ | Xcode 12.4+

Overview

This code project uses the Exposure Notification framework to build a sample app that demonstrates how to notify people when they have come into contact with someone who meets a set of criteria for a case of COVID-19. When using the project as a reference for designing an Exposure Notifications app, you can define the criteria for how the framework determines whether the risk is high enough to report to the user.

The sample app includes code to simulate server responses. When building an Exposure Notifications app based on this project, create a server environment to provide diagnosis keys and exposure criteria, and add code to your app to communicate with this server. If the app you build operates in a country that authenticates medical tests for COVID-19, you may need to include additional network code to communicate with those authentication services.

Exposure Notifications is available on all iOS devices running iOS 13.5 or later. It's also available on iOS 12.5 with some additional setup. The Xcode project has two targets:

- ExposureNotificationApp — Supports iOS 13.5 and later
- ExposureNotificationApp-iOS12 — Supports iOS 12.5, and iOS 13.5 and later

For more information on the architecture and security of the Exposure Notification service, see Privacy-Preserving Contact Tracing. For more information on supporting iOS 12.5 in your Exposure Notifications App, see Supporting Exposure Notifications in iOS 12.5.

Configure the Sample Code Project

Before you run the sample code project in Xcode, make sure:

- Your iOS device is running either iOS 12.5 or 13.5 or later.
- You are running Xcode 12 or later.
- You configure the project with a provisioning profile that includes the Exposure Notification entitlement. To get permission to use this entitlement, see Exposure Notification Entitlement Request.

Authorize Exposure Notifications

Users must explicitly authorize an app to use Exposure Notifications. The `ENManager` class provides information on the user's authorization status and requests authorization. The project has a singleton class called `ExposureManager` that instantiates and manages the life cycle of an `ENManager` object. During `init`, It calls `activate(completionHandler:)` on `ENManager`, then checks the callback to find out if Exposure Notifications is enabled. If it isn't, `ExposureManager` attempts to enable it.

```
    manager.activate { _ in
        // Ensure Exposure Notifications is enabled if the app is authorized. The ap
        // could get into a state where it is authorized, but Exposure Notifications
        // is not enabled, if the user initially denied Exposure Notifications
        // during onboarding, but then flipped on the "COVID-19 Exposure Notificatio
        // in Settings.
        if !self.manager.exposureNotificationEnabled {
            self.manager.setExposureNotificationEnabled(true) { (error) in
                if let error = error {
                    print("Error attempting to enable on launch: \(error.localizedDe
                }
            }
        }
    }
}
```

Store User Data Locally

The app stores information about test results and high-risk exposures in the user defaults directory. The local data is private and stays on the device.

A custom property wrapper transforms data between its native format and a JSON-formatted equivalent, reads and writes data in the user defaults dictionary, and posts notifications to the app when local data changes. The `LocalStore` class manages the user's private data, defined as a series of properties that all use this property wrapper.

```
class LocalStore {

    static let shared = LocalStore()

    @Persisted(userDefaultsKey: "isOnboarded", notificationName: .init("LocalStoreIs
    var isOnboarded: Bool

    @Persisted(userDefaultsKey: "nextDiagnosisKeyFileIndex", notificationName: .init
    var nextDiagnosisKeyFileIndex: Int

    @Persisted(userDefaultsKey: "exposures", notificationName: .init("LocalStoreExpo
    var exposures: [Exposure]

    @Persisted(userDefaultsKey: "dateLastPerformedExposureDetection",
               notificationName: .init("LocalStoreDateLastPerformedExposureDetectior
    var dateLastPerformedExposureDetection: Date?

    @Persisted(userDefaultsKey: "exposureDetectionErrorLocalizedDescription", notifi
        .init("LocalStoreExposureDetectionErrorLocalizedDescriptionDidChange"), defa
    var exposureDetectionErrorLocalizedDescription: String?

    @Persisted(userDefaultsKey: "testResults", notificationName: .init("LocalStoreTe
    var testResults: [UUID: TestResult]
}
```

The app defines its own data structures for any data it persists. For example, a test result records the date the user took the test, and whether the user shared this data with the server. This information is used to populate the user interface.

```
struct TestResult: Codable {
    var id: UUID                // A unique identifier for this test result
    var isAdded: Bool           // Whether the user completed the add positive diagr
    var dateAdministered: Date  // The date the test was administered
    var isShared: Bool          // Whether diagnosis keys were shared with the Healt
}
```

Share Diagnosis Keys with the Server

A user with a diagnosis for COVID-19 can upload *diagnosis keys* to the server. Each instance of the app periodically downloads diagnosis keys to search the device's private interaction data for matching interactions.

This project simulates a remote server with which the app communicates. There is a single `Server` object in the app that stores the received diagnosis keys and provides them on demand. The sample server does not partition the data by region. It maintains a single list of keys, and provides the entire list upon request.

```
// Replace this class with your own class that communicates with your server.
class Server {

    static let shared = Server()

    // For testing purposes, this object stores all of the TEKs it receives locally
    // In a real implementation, these would be stored on a remote server
    @Persisted(userDefaultsKey: "diagnosisKeys", notificationName: .init("ServerDiag
    var diagnosisKeys: [CodableDiagnosisKey]
```

As with the local store, this local server stores the data in JSON format, using the same `Persisted` property wrapper.

Ask Users to Share COVID-19 Indicators

The sample app demonstrates a strategy in which a recognized medical authority tested the user and found positive COVID-19 indicators. The sample app provides a way for users to enter an authentication code, but doesn't submit this data to an authentication service, so all codes automatically pass.

When the user provides information about a positive test result, the app records the test result in the local store and asks the user to share it. To share the result, the app needs to get a list of diagnosis keys and send the list to the server. To get the keys, the app calls the singleton `ENManager` object's `getDiagnosisKeys(completionHandler:)` method, as shown in the code below.

```
func getAndPostDiagnosisKeys(testResult: TestResult, completion: @escaping (Error?)
    manager.getDiagnosisKeys { temporaryExposureKeys, error in
        if let error = error {
            completion(error)
        } else {
```

```
            guard let temporaryExposureKeys = temporaryExposureKeys else {
                print("No exposure keys, aborting key share")
                return
            }

            // In this sample app, transmissionRiskLevel isn't set for any of the di
            // use information accumulated in testResult to determine a transmissior
            Server.shared.postDiagnosisKeys(temporaryExposureKeys) { error in
                completion(error)
            }
        }
    }
}
```

Each time the app calls this method, the user must authorize the transaction. The framework then returns the list of keys to the app. The app sends those keys as-is to the server and then updates the test record to indicate that it was shared.

The sample app's server implementation appends the keys onto a list it maintains, skipping any keys that are already there. It stores the keys sequentially so that the app can request just the keys it hasn't received before.

```
func postDiagnosisKeys(_ diagnosisKeys: [ENTemporaryExposureKey], completion: (Erro

    // Convert keys to something that can be encoded to JSON and upload them.
    let codableDiagnosisKeys = diagnosisKeys.compactMap { diagnosisKey -> CodableDia
        return CodableDiagnosisKey(keyData: diagnosisKey.keyData,
                                   rollingPeriod: diagnosisKey.rollingPeriod,
                                   rollingStartNumber: diagnosisKey.rollingStartNumb
                                   transmissionRiskLevel: diagnosisKey.transmissionR
    }

    // In a real implementation, these keys would be uploaded with URLSession instea
    // Your server needs to handle de-duplicating keys.
    for codableDiagnosisKey in codableDiagnosisKeys where !self.diagnosisKeys.contai
        self.diagnosisKeys.append(codableDiagnosisKey)
    }
    completion(nil)
}
```

Ask Users to Preauthorize Key Release at the Time of the Test

Starting with iOS 14.4, the Exposure Notification framework allows apps to ask users for permission to release temporary exposure keys when they take a COVID-19 diagnostic test. This authorization lasts for up to five days and should be requested only if the app determines that the user is about to take a test and the app has a way to determine the results. To request preauthorization, the app calls `preAuthorizeDiagnosisKeys(completionHandler:)`.

```
@available(iOS 14.4, *)
func preAuthorizeKeys(
    completion: @escaping (Error?) -> Void) {
    manager.preAuthorizeDiagnosisKeys { (error) in
        if let error = error {
            print("Error pre-authorizing keys: \(error)")
            completion(error)
            return
        }
        print("Successfully pre-authorized keys")
        completion(nil)
    }
}
```

Within five days of being granted permission, if the app determines that the user has received a positive test result, it can request the preauthorized keys and submit them to the key server. `ENManager` sends keys to the app using a completion handler property called `diagnosisKeysAvailableHandler`, which the app sets before requesting the keys:

```
// This handler receives preauthorized keys. Once the handler is called,
// the preauthorization expires, so the handler should only be called
// once per preauthorization request. If the user doesn't authorize
// release, this handler isn't called.
manager.diagnosisKeysAvailableHandler = { (keys) in
    Server.shared.postDiagnosisKeys(keys) { (error) in
        if let error = error {
            print("Error posting pre-authorized diagnosis keys: \(error)")
        }
        completion(error)
    }
}
```

After setting the `diagnosisKeysAvailableHandler` property, the app requests the keys by calling `requestPreAuthorizedDiagnosisKeys(completionHandler:)`. This call returns an error if the user doesn't authorize release or if more than five days pass after authorization. Otherwise, the completion handler is called with the keys. When the sample app releases the keys, it notifies the user that because they've had a positive test result, their keys are being shared pursuant to their prior authorization.

```
// This call requests preauthorized keys. The request fails if the
// user doesn't authorize release or if more than five days pass after
// authorization. If requestPreAuthorizedDiagnosisKeys(:) has already
// been called since the last time the user preauthorized, the call
// doesn't fail but also doesn't return any keys.
manager.requestPreAuthorizedDiagnosisKeys { (error) in
    if let error = error {
        print("Error retrieving pre-authorized diganosis keys: \(error)")
        completion(error)
    }
}
```

Important

PHAs must notify users that they've tested positive *before* the app calls `requestPreAuthorizedDiagnosisKeys(completionHandler:)`. They must also notify users that their keys are being submitted to the PHA's key server.

Detect Exposure Notifications API Version at Runtime

The Exposure Notifications APIs are available in the following distinct versions that implement slightly different versions of the detection algorithm:

- Version 1 — Devices running iOS 13.5 or iOS 13.6 support the version 1 API only. To prevent runtime errors, Exposure Notifications apps for iOS 13.5 and iOS 13.6 must fall back to this version.
- Version 2 — When available, apps should use this version of the API.

The sample app includes a utility function to determine which API versions are available on the current device:

```
func ENManagerIsAvailable() -> Bool {
    return NSClassFromString("ENManager") != nil
}
```

```
enum SupportedENAPIVersion {
    case version2
    case version1
    case unsupported
}

func getSupportedExposureNotificationsVersion() -> SupportedENAPIVersion {
    if #available(iOS 13.7, *) {
        return .version2
    } else if #available(iOS 13.5, *) {
        return .version1
    } else if ENManagerIsAvailable() {
        return .version2
    } else {
        return .unsupported
    }
}
```

The main difference between using the two API versions is the method called to detect exposures. When running on devices that only support version 1, the sample app uses `getExposureInfo(summary:userExplanation:)` to evaluate diagnosis keys for potential exposures. When running on devices that support version 2 of the API, it uses `getExposureWIndows(summary:completionHandler:)` instead.

```
// Handles getting exposures using the version 2 API in iOS 13.7+ and
// in iOS 12.5
func getExposuresV2(_ summary: ENExposureDetectionSummary) {
    self.manager.getExposureWindows(summary: summary) { windows, error in
        if let error = error {
            finish(.failure(error))
            return
        }
        let allWindows = windows!.map { window in
            Exposure(date: window.date)
        }
        finish(.success((allWindows, nextDiagnosisKeyFileIndex + localURLs.count)))
    }
}

// Handles getting exposures using the version 1 API used in iOS 13.5 and iOS 13.6
@available(iOS 13.5, *)
```

```
func getExposuresV1(_ summary: ENExposureDetectionSummary) {
    let userExplanation = NSLocalizedString("USER_NOTIFICATION_EXPLANATION", comment
    ExposureManager.shared.manager.getExposureInfo(summary: summary,
                                                   userExplanation: userExplanation)
        if let error = error {
            finish(.failure(error))
            return
        }
        let newExposures = exposures!.map { exposure in
            Exposure(date: exposure.date)
        }
        finish(.success((newExposures, nextDiagnosisKeyFileIndex + localURLs.count))
    }
}
```

Check for Exposures in iOS 13.5+

The ExposureNotificationApp target demonstrates how to create a background task in iOS 13.5 or later to periodically download new keys and check whether the user may have been exposed to an individual with COVID-19.

The app's `Info.plist` file declares a background task named `com.example.apple-samplecode.ExposureNotificationSampleApp.exposure-notification`. The Background Task framework automatically detects apps that contain the Exposure Notification entitlement and a background task that ends in `exposure-notification`. The operating system automatically launches these apps when they aren't running and guarantees them more background time to ensure that the app can test and report results promptly.

```
func scheduleBackgroundTaskIfNeeded() {
    if #available(iOS 13.5, *) {
        guard ENManager.authorizationStatus == .authorized else { return }
        let taskRequest = BGProcessingTaskRequest(identifier: backgroundTaskIdentifi
        taskRequest.requiresNetworkConnectivity = true
        do {
            try BGTaskScheduler.shared.submit(taskRequest)
        } catch {
            print("Unable to schedule background task: \(error)")
        }
    }
}
```

First, the background task provides a handler in case it runs out of time. Then it calls the app's `detectExposures` method to test for exposures. Finally, it schedules the next time the system should execute the background task.

```
func createBackgroundTaskIfNeeded() {
    if #available(iOS 13.0, *) {
        BGTaskScheduler.shared.register(forTaskWithIdentifier: backgroundTaskIdentif

            // Notify the user if Bluetooth is off
            ExposureManager.shared.showBluetoothOffUserNotificationIfNeeded()

            // Perform the exposure detection
            let progress = ExposureManager.shared.detectExposures { success in
                task.setTaskCompleted(success: success)
            }

            // Handle running out of time
            task.expirationHandler = {
                progress.cancel()
                LocalStore.shared.exposureDetectionErrorLocalizedDescription = NSLoc
            }

            // Schedule the next background task
            scheduleBackgroundTaskIfNeeded()
        }
    }
}
```

Check for Exposures in iOS 12.5

To support iOS 12.5, apps must register an activity handler with ENManager's `setLaunch ActivityHandler()` instead of creating a background task, but only when running on iOS 12.5. This step is necessary because Background Tasks do not exist in iOS 12.5. Instead, `ENManager` provides apps that register an activity handler with 3.5 minutes of background processing at least once per day.

```
if #available(iOS 13.5, *) {
    // In iOS 13.5 and later, the Background Tasks framework is available,
    // so create and schedule a background task for downloading keys and
    // detecting exposures
    createBackgroundTaskIfNeeded()
```

```
    scheduleBackgroundTaskIfNeeded()
} else if ENManagerIsAvailable() {
    // If `ENManager` exists, and the iOS version is earlier than 13.5,
    // the app is running on iOS 12.5, where the Background Tasks
    // framework is unavailable. Specify an EN activity handler here, which
    // allows the app to receive background time for downloading keys
    // and looking for exposures when background tasks aren't available.
    // Apps should should call this method before calling activate().
    manager.setLaunchActivityHandler { (activityFlags) in
        // ENManager gives apps that register an activity handler
        // in iOS 12.5 up to 3.5 minutes of background time at
        // least once per day. In iOS 13 and later, registering an
        // activity handler does nothing.
        if activityFlags.contains(.periodicRun) {
            print("Periodic activity callback called (iOS 12.5)")
            _ = ExposureManager.shared.detectExposures()
        }
    }
}
```

The remaining sections describe how the app obtains the set of diagnosis keys and submits them to the framework for evaluation.

Download Diagnosis Keys

The app downloads diagnosis keys from the server to pass to the framework, starting with the first key the app hasn't downloaded before. This design ensures that the app checks each diagnosis key only once on any given device.

The app needs to provide signed key files to the framework. The app asks the server for the URLs of any key files that the server generated after the last file that the app checked. After receiving the URLs from the server, the app uses a dispatch group to download the files to the device.

```
let nextDiagnosisKeyFileIndex = LocalStore.shared.nextDiagnosisKeyFileIndex

Server.shared.getDiagnosisKeyFileURLs(startingAt: nextDiagnosisKeyFileIndex) { resul

    let dispatchGroup = DispatchGroup()
    var localURLResults = [Result<[URL], Error>]()

    switch result {
    case let .success(remoteURLs):
```

```
        for remoteURL in remoteURLs {
            dispatchGroup.enter()
            Server.shared.downloadDiagnosisKeyFile(at: remoteURL) { result in
                localURLResults.append(result)
                dispatchGroup.leave()
            }
        }

    case let .failure(error):
        finish(.failure(error))
    }
```

Finally, the app creates an array of the local URLs for the downloaded files.

```
dispatchGroup.notify(queue: .main) {
    for result in localURLResults {
        switch result {
        case let .success(urls):
            localURLs.append(contentsOf: urls)
        case let .failure(error):
            finish(.failure(error))
            return
        }
    }
```

Configure Criteria to Estimate Risk

The framework will compare locally saved interaction data against the diagnosis keys provided by the app. When the framework finds a match, it calculates a risk score for that interaction based on a number of different factors, such as when the interaction took place and how long the devices were in proximity to each other.

To provide specific guidance to the framework about how risk should be evaluated, the app creates an `ENExposureConfiguration` object. The app requests the criteria from the `Server` object, which creates and returns an `ENExposureConfiguration` object as shown below. The sample configuration has placeholder data that evaluates any interaction as risky, so the framework returns all interactions that match the diagnosis keys.

```
func getExposureConfiguration(completion: (Result<ENExposureConfiguration, Error>) -

    let dataFromServer = """
    {
```

```
    "immediateDurationWeight":100,
    "nearDurationWeight":100,
    "mediumDurationWeight":100,
    "otherDurationWeight":100,
    "infectiousnessForDaysSinceOnsetOfSymptoms":{
        "unknown":1,
        "-14":1,
        "-13":1,
        "-12":1,
        "-11":1,
        "-10":1,
        "-9":1,
        "-8":1,
        "-7":1,
        "-6":1,
        "-5":1,
        "-4":1,
        "-3":1,
        "-2":1,
        "-1":1,
        "0":1,
        "1":1,
        "2":1,
        "3":1,
        "4":1,
        "5":1,
        "6":1,
        "7":1,
        "8":1,
        "9":1,
        "10":1,
        "11":1,
        "12":1,
        "13":1,
        "14":1
    },
    "infectiousnessStandardWeight":100,
    "infectiousnessHighWeight":100,
    "reportTypeConfirmedTestWeight":100,
    "reportTypeConfirmedClinicalDiagnosisWeight":100,
    "reportTypeSelfReportedWeight":100,
    "reportTypeRecursiveWeight":100,
    "reportTypeNoneMap":1,
```

```
    "minimumRiskScore":0,
    "attenuationDurationThresholds":[50, 70],
    "attenuationLevelValues":[1, 2, 3, 4, 5, 6, 7, 8],
    "daysSinceLastExposureLevelValues":[1, 2, 3, 4, 5, 6, 7, 8],
    "durationLevelValues":[1, 2, 3, 4, 5, 6, 7, 8],
    "transmissionRiskLevelValues":[1, 2, 3, 4, 5, 6, 7, 8]
    }
    """.data(using: .utf8)!

    do {
        let codableExposureConfiguration = try JSONDecoder().decode(CodableExposureC
        let exposureConfiguration = ENExposureConfiguration()
        if ENManagerIsAvailable() {
            exposureConfiguration.immediateDurationWeight = codableExposureConfigura
            exposureConfiguration.nearDurationWeight = codableExposureConfiguration.
            exposureConfiguration.mediumDurationWeight = codableExposureConfiguratio
            exposureConfiguration.otherDurationWeight = codableExposureConfiguration
            var infectiousnessForDaysSinceOnsetOfSymptoms = [Int: Int]()
            for (stringDay, infectiousness) in codableExposureConfiguration.infectio
                if stringDay == "unknown" {
                    if #available(iOS 14.0, *) {
                        infectiousnessForDaysSinceOnsetOfSymptoms[ENDaysSinceOnsetOf
                    } else {
                        // ENDaysSinceOnsetOfSymptomsUnknown is not available
                        // in earlier versions of iOS; use an equivalent value
                        infectiousnessForDaysSinceOnsetOfSymptoms[NSIntegerMax] = in
                    }
                } else if let day = Int(stringDay) {
                    infectiousnessForDaysSinceOnsetOfSymptoms[day] = infectiousness
                }
            }
            exposureConfiguration.infectiousnessForDaysSinceOnsetOfSymptoms = infect
            exposureConfiguration.infectiousnessStandardWeight = codableExposureConf
            exposureConfiguration.infectiousnessHighWeight = codableExposureConfigur
            exposureConfiguration.reportTypeConfirmedTestWeight = codableExposureCon
            exposureConfiguration.reportTypeConfirmedClinicalDiagnosisWeight = codab
            exposureConfiguration.reportTypeSelfReportedWeight = codableExposureConf
            exposureConfiguration.reportTypeRecursiveWeight = codableExposureConfigu
            if let reportTypeNoneMap = ENDiagnosisReportType(rawValue: UInt32(codab
                exposureConfiguration.reportTypeNoneMap = reportTypeNoneMap
            }
        }
        exposureConfiguration.minimumRiskScore = codableExposureConfiguration.minimu
```

```
        exposureConfiguration.attenuationLevelValues = codableExposureConfiguration.
        exposureConfiguration.daysSinceLastExposureLevelValues = codableExposureConf
        exposureConfiguration.durationLevelValues = codableExposureConfiguration.dur
        exposureConfiguration.transmissionRiskLevelValues = codableExposureConfigura
        exposureConfiguration.metadata = ["attenuationDurationThresholds": codableEx
        completion(.success(exposureConfiguration))
    } catch {
        completion(.failure(error))
    }
```

Submit Diagnosis Keys to the Framework

After downloading the key files, the app performs the search for exposures using a series of asynchronous steps. First, the app requests the criteria from the server, which calls into the code shown in the Configure Criteria to Estimate Risk section above. Then the app calls the `ENManager` objects's `detectExposures(configuration:diagnosisKeyURLs:completionHandler:)` method, passing the criteria and the URLs for the downloaded key files. This method returns a summary of the search results.

The code then passes off the downloaded keys to one of two different methods based on which API version is available on the device.

```
Server.shared.getExposureConfiguration { result in
    switch result {
    case let .success(configuration):
        ExposureManager.shared.manager.detectExposures(configuration: configuration,
            if let error = error {
                finish(.failure(error))
                return
            }
            if #available(iOS 13.7, *) {
                getExposuresV2(summary!)
            } else if #available(iOS 13.5, *) {
                getExposuresV1(summary!)
            } else if ENManagerIsAvailable() {
                getExposuresV2(summary!)
            } else {
                print("Exposure Notifications not supported on this version of iOS."
            }
        }

    case let .failure(error):
```

```
            finish(.failure(error))
        }
    }
```

Finally, the app calls its `finish` method to complete the search. The `finish` method updates the local store with the new data, including any exposures, the date and time the app executed the search, and the index for the key file to check next time.

See Also

Essentials

Supporting Exposure Notifications Express

Configure servers to notify users of potential exposures to COVID-19 without an app.

Setting Up a Key Server

Ensure that your server meets the requirements for supporting Exposure Notifications.

`class ENManager`

A class that manages exposure notifications.

`ENDeveloperRegion`

A string that specifies the region that the app supports.

`ENAPIVersion`

A number that specifies the version of the API to use.

Changing Configuration Values Using the Server-to-Server API

Update Exposure Notifications configuration values from a Public Health Authority's server.

Testing Exposure Notifications Apps in iOS 13.7 and Later

Perform end-to-end validation of Exposure Notifications apps on a device by manually loading configuration files.

Supporting Exposure Notifications in iOS 12.5

Prepare your Exposure Notifications app to run on a previous version of iOS.