class SimpleBLEManager: NSObject, CBCentralManagerDelegate, CBPeripheralDelegate {

var targetPeripheral: CBPeripheral?  
 lazy var centralManager: CBCentralManager = {

return CBCentralManager(delegate: self, queue: nil)

}() // this defers creation of centralManager until runtime

func centralManagerDidUpdateState(\_ central: CBCentralManager) {

if central.state == .poweredOn {

centralManager.scanForPeripherals(withServices: nil, options: nil)

}

// ...

}

func centralManager(\_ central: CBCentralManager,

didDiscover peripheral: CBPeripheral,

advertisementData: [String : Any], rssi RSSI: NSNumber) {

targetPeripheral = peripheral

centralManager.stopScan()

centralManager.connect(peripheral)

}

func centralManager(\_ central: CBCentralManager,

didConnect peripheral: CBPeripheral) {

print("Connected to \(peripheral.name ?? "device")!")

}

}

let simpleBLEManager = SimpleBLEManager() // usage

import MapKit

import CoreLocation

class MyViewController: UIViewController,

CLLocationManagerDelegate,

MKMapViewDelegate {

@IBOutlet weak var mapView: MKMapView!

var locationManager: CLLocationManager!

override func viewDidLoad() {

super.viewDidLoad()

mapView.showsUserLocation = true

mapView.delegate = self

locationManager = CLLocationManager()

locationManager.delegate = self

locationManager.requestWhenInUseAuthorization()

}

func locationManager(\_ manager: CLLocationManager,

didChangeAuthorization status: CLAuthorizationStatus) {

if manager.authorizationStatus == .authorizedWhenInUse {

locationManager.startUpdatingLocation()

}

}

func locationManager(\_ manager: CLLocationManager,

didUpdateLocations locations: [CLLocation]) {

guard let location = locations.last else { return }

// Move to user's location on the map

let region = MKCoordinateRegion(center: location.coordinate,

latitudinalMeters: 1000,

longitudinalMeters: 1000)

mapView.setRegion(region, animated: true)

}

}

import Combine

class TimerViewModel: ObservableObject {

// a property wrapper provided by Combine:

@Published var value: Int = 0

// a type that can be used to cancel a subscription:

private var cancellable: AnyCancellable?

init() {

cancellable = Timer.publish(every: 2, on: .main, in: .common)

// This starts the timer as soon as a subscriber is attached:

.autoconnect()

// This is a Combine method that "subscribes" to events:

.sink { [weak self] \_ in

guard let self = self else { return } // Ensure `self` is still valid

self.value += 2

}

}

deinit { cancellable?.cancel() }

}

struct TimerView: View {

// a SwiftUI property wrapper:

@ObservedObject var viewModel = TimerViewModel()

var body: some View {

Text("Timer Value: \(viewModel.value)")

}

}

import CoreNFC

class NFCViewController: UIViewController, NFCNDEFReaderSessionDelegate {

var nfcSession: NFCNDEFReaderSession?

func startNFCSession() {

nfcSession = NFCNDEFReaderSession(delegate: self,

queue: nil, invalidateAfterFirstRead: false)

nfcSession?.begin()

}

func readerSession(\_ session: NFCNDEFReaderSession,

didInvalidateWithError error: Error) {

// handle session invalidation

}

func readerSession(\_ session: NFCNDEFReaderSession,

didDetectNDEFs messages: [NFCNDEFMessage]) {

// handle initial NFC detection

}

// Delegate method for tag detection

func readerSession(\_ session: NFCNDEFReaderSession,

didDetect tags: [NFCNDEFTag]) {

if let tag = tags.first {

session.connect(to: tag) { (error: Error?) in

// Handle connection error

}

}

}

// ...

}

extension NFCViewController {

// Delegate method for reading NDEF message

func readerSession(\_ session: NFCNDEFReaderSession,

didDetectNDEFs messages: [NFCNDEFMessage]) {

for message in messages {

for record in message.records {

// Process NDEF records

}

}

session.invalidate()

}

// Delegate method for writing NDEF message

func readerSession(\_ session: NFCNDEFReaderSession,

didWriteNDEFMessage message: NFCNDEFMessage) {

// Handle successful write

session.invalidate()

}

}

import Intents

protocol OrderRideIntentHandling: AnyObject {

func handle(

intent: OrderRideIntent,

completion: @escaping (OrderRideIntentResponse) -> Void)

}

class OrderRideIntent: INIntent {

var location: CLPlacemark?

}

class OrderRideIntentResponse: INIntentResponse {

enum Code: Int {

case success

case failure

}

let code: Code

let activity: NSUserActivity?

init(code: Code, activity: NSUserActivity?) {

self.code = code

self.activity = activity

super.init()

}

required init?(coder: NSCoder) {

guard let code = Code(rawValue: coder.decodeInteger(forKey: "code")),

let activity = coder.decodeObject(forKey: "activity")

as? NSUserActivity

else {

return nil

}

self.code = code

self.activity = activity

super.init(coder: coder)

}

}

class CustomIntentHandler: INExtension, OrderRideIntentHandling {

func handle(intent: OrderRideIntent,

completion: @escaping (OrderRideIntentResponse) -> Void) {

// Implement logic to handle the Intent & response here ...

let response = OrderRideIntentResponse(code: .success, activity: nil)

completion(response)

}

}

import HealthKit

let healthStore = HKHealthStore()

let stepCountType = HKQuantityType.quantityType(forIdentifier: .stepCount)!

func requestHealthKitAuthorization() {

if !HKHealthStore.isHealthDataAvailable() {

return

}

healthStore.requestAuthorization(

toShare: [stepCountType],

read: [stepCountType]) { (success, error) in

if success {

// You can now access step count data

} else {

// Authorization failed

}

}

}

func readStepCount() {

// Define a query to retrieve step count data

let query = HKSampleQuery(

sampleType: stepCountType,

predicate: nil,

limit: HKObjectQueryNoLimit,

sortDescriptors: nil) { (query, results, error) in

if let queryResults = results as? [HKQuantitySample] {

for result in queryResults {

let steps = result.quantity.doubleValue(

for: HKUnit.count())

print("Steps: \(steps)")

}

} else {

// no step count data is available or an error occurred

}

}

healthStore.execute(query)

}

func writeStepCount(steps: Double) {

// Create a quantity for the number of steps

let stepsQuantity = HKQuantity(unit: HKUnit.count(), doubleValue: steps)

// Create an HKQuantitySample object

let stepCountSample = HKQuantitySample(type: stepCountType,

quantity: stepsQuantity,

start: Date(),

end: Date())

// Save the data to HealthKit

healthStore.save(stepCountSample) { (success, error) in

if success {

print("Step count saved.")

} else {

// Step count data couldn't be saved

}

}

}