I 💕 SWIFT CONCURRENCY

BY TUNDSDEV

A bit about me

Lead Dev @ Bally's Interactive

Currently building amazing experiences within the Sports Product Studio

I'm a Content Creator

I currently teach iOS development mainly Swift & SwiftUI on my **YouTube** channel **tundsdev**

I love Anime

When I'm not coding, to relax I love watching Anime & hope to visit Japan next year

I live for random

I love travelling or exploring new things, just like you can see me landing in this ball pit like a potato


YOU GET A CODE SAMPLE

WHAT IS SWIFT CONCURRENCY?

New language updates introduced at WWDC21

🎯 Goals

Easy to understand

Less closures, less problems. Easier to read compared to closure based syntax

Simple to write

Writing asynchronous code is easier to write and maintain

Less race conditions

Mutating data on different threads is safer

Free up threads

Makes better use of threads for long running tasks

WHAT IS ASYNCHRONOUS CODE?

👀

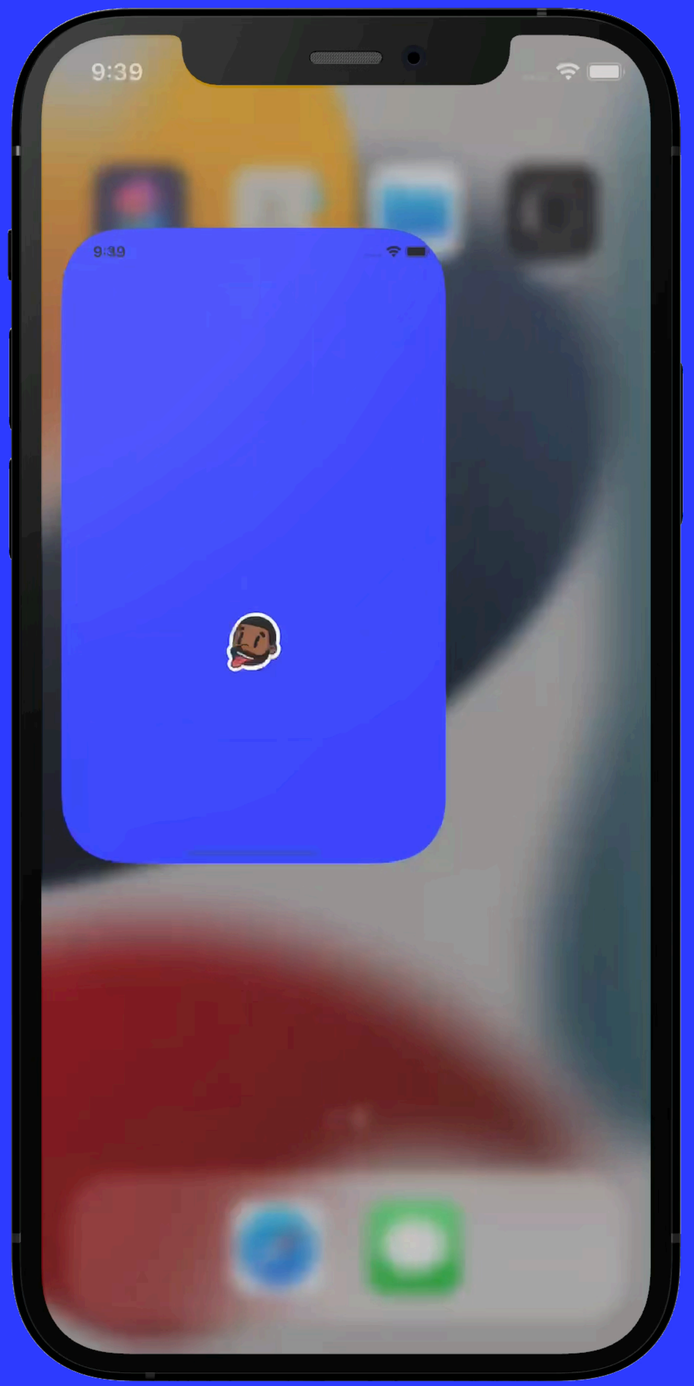
The difference

Synchronous

Single thread is used so only one task will be executed at a time, blocking the main thread

Asynchronous

Multi threaded so tasks can be executed in parallel at the same time, non blocking the main thread.


9:39
9:39


Tarihi
Sultanahmet Köftecisi
Selim Usta
Kuruluş
1920
Since
1920
Tarihi
Sultanahmet
Köftecisi
Selim Usta
STAR
HOLİDAY HOTEL
SAHURA KADAR AÇIĞIZ
Köfte Ekmek
Ayran
ÇORBA
1 PAKET

Async/Await

```swift
func request<T: Decodable>(_ absoluteURL: String,
                           type: T.Type,
                           completion: @escaping (Result<T, Error>) -> Void) {

    guard let url = URL(string: absoluteURL) else {
        completion(.failure(NetworkingError.invalidUrl))
        return
    }

    let request = URLRequest(url: url)

    let dataTask = URLSession.shared.dataTask(with: request) { data, response, error in

        if error != nil {
            completion(.failure(NetworkingError.custom(error: error!)))
            return
        }

        guard let response = response as? HTTPURLResponse,
                (200...300) ~= response.statusCode else {
            completion(.failure(NetworkingError.invalidStatusCode))
            return
        }
```

Async/Await

```
        if error != nil {
            completion(.failure(NetworkingError.custom(error: error!)))
            return
        }

        guard let response = response as? HTTPURLResponse,
                (200...300) ~= response.statusCode else {
            completion(.failure(NetworkingError.invalidStatusCode))
            return
        }

        guard let data = data else {
            completion(.failure(NetworkingError.invalidData))
            return
        }

        do {
            let decoder = JSONDecoder()
            decoder.keyDecodingStrategy = .convertFromSnakeCase
            let res = try decoder.decode(T.self, from: data)
            completion(.success(res))

        } catch {

        }
    }

    dataTask.resume()
}
```

Async/Await

```swift
func request<T: Decodable>(_ absoluteURL: String,
                           type: T.Type,
                           completion: @escaping (Result<T, Error>) -> Void) {

    guard let url = URL(string: absoluteURL) else {
        completion(.failure(NetworkingError.invalidUrl))
        return
    }

    let request = URLRequest(url: url)

    let dataTask = URLSession.shared.dataTask(with: request) { data, response, error in

        if error != nil {
            completion(.failure(NetworkingError.custom(error: error!)))
            return
        }

        guard let response = response as? HTTPURLResponse,
                (200...300) ~= response.statusCode else {
            completion(.failure(NetworkingError.invalidStatusCode))
            return
        }

        guard let data = data else {
            completion(.failure(NetworkingError.invalidData))
            return
        }

        do {
            let decoder = JSONDecoder()
            decoder.keyDecodingStrategy = .convertFromSnakeCase
            let res = try decoder.decode(T.self, from: data)
            completion(.success(res))

        } catch {

        }
    }

    dataTask.resume()
}
```

Difficult to read

A lot to write

Easy to forget

Async/Await

```
    let dataTask = URLSession.shared.dataTask(with: request) { data, response, error in

        if error != nil {
            completion(.failure(NetworkingError.custom(error: error!)))
            return
        }

        guard let response = response as? HTTPURLResponse,
                (200...300) ~= response.statusCode else {
            completion(.failure(NetworkingError.invalidStatusCode))
            return
        }

        guard let data = data else {
            completion(.failure(NetworkingError.invalidData))
            return
        }

        do {
            let decoder = JSONDecoder()
            decoder.keyDecodingStrategy = .convertFromSnakeCase
            let res = try decoder.decode(T.self, from: data)
            completion(.success(res))

        } catch {

        }
    }

    dataTask.resume()
}
```

Difficult to read

A lot to write

Easy to forget

Async/Await

```
    let dataTask = URLSession.shared.dataTask(with: request) { data, response, error in

        if error != nil {
            completion(.failure(NetworkingError.custom(error: error!)))
            return
        }

        guard let response = response as? HTTPURLResponse,
                (200...300) ~= response.statusCode else {
            completion(.failure(NetworkingError.invalidStatusCode))
            return
        }

        guard let data = data else {
            completion(.failure(NetworkingError.invalidData))
            return
        }

        do {
            let decoder = JSONDecoder()
            decoder.keyDecodingStrategy = .convertFromSnakeCase
            let res = try decoder.decode(T.self, from: data)
            completion(.success(res))

        } catch {

        }
    }

    dataTask.resume()
}
```

Difficult to read

A lot to write

Easy to forget

Async/Await

```swift
func request<T: Codable>(session: URLSession = .shared,
                         _ endpoint: Endpoint,
                         type: T.Type) async throws -> T {

    guard let url = endpoint.url else {
        throw NetworkingError.invalidUrl
    }

    let request = buildRequest(from: url, methodType: endpoint.methodType)

    let (data, response) = try await session.data(for: request)

    guard let response = response as? HTTPURLResponse,
          (200...300) ~= response.statusCode else {
        let statusCode = (response as! HTTPURLResponse).statusCode
        throw NetworkingError.invalidStatusCode(statusCode: statusCode)
    }

    let decoder = JSONDecoder()
    decoder.keyDecodingStrategy = .convertFromSnakeCase
    let res = try decoder.decode(T.self, from: data)

    return res
}
```

Easy to read

Fewer LOC

Case paths handled

No need for .resume()

Async/Await

```swift
func request<T: Codable>(session: URLSession = .shared,
                         _ endpoint: Endpoint,
                         type: T.Type) async throws -> T {

    guard let url = endpoint.url else {
        throw NetworkingError.invalidUrl
    }

    let request = buildRequest(from: url, methodType: endpoint.methodType)

    let (data, response) = try await session.data(for: request)

    guard let response = response as? HTTPURLResponse,
          (200...300) ~= response.statusCode else {
        let statusCode = (response as! HTTPURLResponse).statusCode
        throw NetworkingError.invalidStatusCode(statusCode: statusCode)
    }

    let decoder = JSONDecoder()
    decoder.keyDecodingStrategy = .convertFromSnakeCase
    let res = try decoder.decode(T.self, from: data)

    return res
}
```

async keyword marks a function as **asynchronous**

await keyword marks a **suspension** point

try/throws keywords allow errors to be thrown from the function

No closures needed functions can be void or return a value

Async/Await

```swift
func request<T: Codable>(session: URLSession = .shared,
                         _ endpoint: Endpoint,
                         type: T.Type) async throws -> T {

    guard let url = endpoint.url else {
        throw NetworkingError.invalidUrl
    }

    let request = buildRequest(from: url, methodType: endpoint.methodType)

    let (data, response) = try await session.data(for: request)

    guard let response = response as? HTTPURLResponse,
          (200...300) ~= response.statusCode else {
        let statusCode = (response as! HTTPURLResponse).statusCode
        throw NetworkingError.invalidStatusCode(statusCode: statusCode)
    }

    let decoder = JSONDecoder()
    decoder.keyDecodingStrategy = .convertFromSnakeCase
    let res = try decoder.decode(T.self, from: data)

    return res
}
```

async keyword marks a function as **asynchronous**

await keyword marks a **suspension** point

try/throws keywords allow errors to be thrown from the function

No closures needed functions can be void or return a value

Async/Await

```swift
func request<T: Codable>(session: URLSession = .shared,
                         _ endpoint: Endpoint,
                         type: T.Type) async throws -> T {

    guard let url = endpoint.url else {
        throw NetworkingError.invalidUrl
    }

    let request = buildRequest(from: url, methodType: endpoint.methodType)

    let (data, response) = try await session.data(for: request)

    guard let response = response as? HTTPURLResponse,
          (200...300) ~= response.statusCode else {
        let statusCode = (response as! HTTPURLResponse).statusCode
        throw NetworkingError.invalidStatusCode(statusCode: statusCode)
    }

    let decoder = JSONDecoder()
    decoder.keyDecodingStrategy = .convertFromSnakeCase
    let res = try decoder.decode(T.self, from: data)

    return res
}
```

async keyword marks a function as **asynchronous**

await keyword marks a **suspension** point

try/**throws** keywords allow errors to be thrown from the function

No closures needed functions can be void or return a value

Async/Await

```swift
func request<T: Codable>(session: URLSession = .shared,
                         _ endpoint: Endpoint,
                         type: T.Type) async throws -> T {
    guard let url = endpoint.url else {
        throw NetworkingError.invalidUrl
    }

    let request = buildRequest(from: url, methodType: endpoint.methodType)

    let (data, response) = try await session.data(for: request)

    guard let response = response as? HTTPURLResponse,
          (200...300) ~= response.statusCode else {
        let statusCode = (response as! HTTPURLResponse).statusCode
        throw NetworkingError.invalidStatusCode(statusCode: statusCode)
    }

    let decoder = JSONDecoder()
    decoder.keyDecodingStrategy = .convertFromSnakeCase
    let res = try decoder.decode(T.self, from: data)

    return res
}
```

async keyword marks a function as **asynchronous**

await keyword marks a **suspension** point

try/throws keywords allow errors to be thrown from the function

No closures needed functions can be void or return a value

Updating the UI

```swift
final class MusicViewModel: ObservableObject {
    @Published private(set) var songs: [Song] = []

    func fetchSongs() async {
        self.songs = try? await NetworkingManager.shared.request(session: .shared,
                                                                 .songs,
                                                                 type: [SongsResponse].self) ?? []

    }
}
```

Updating the UI

```swift
final class MusicViewModel: ObservableObject {
    @Published private(set) var songs: [Song] = []

    func fetchSongs() async {
        self.songs = try? await NetworkingManager.shared.request(session: .shared,
                                                                 .songs,
                                                                 type: [SongsResponse].self) ?? []
    }
}
```

⚠ Publishing changes from background threads is not allowed; make sure to publish values from the main thread (via operators like receive(on:)) on model updates.

Updating the UI

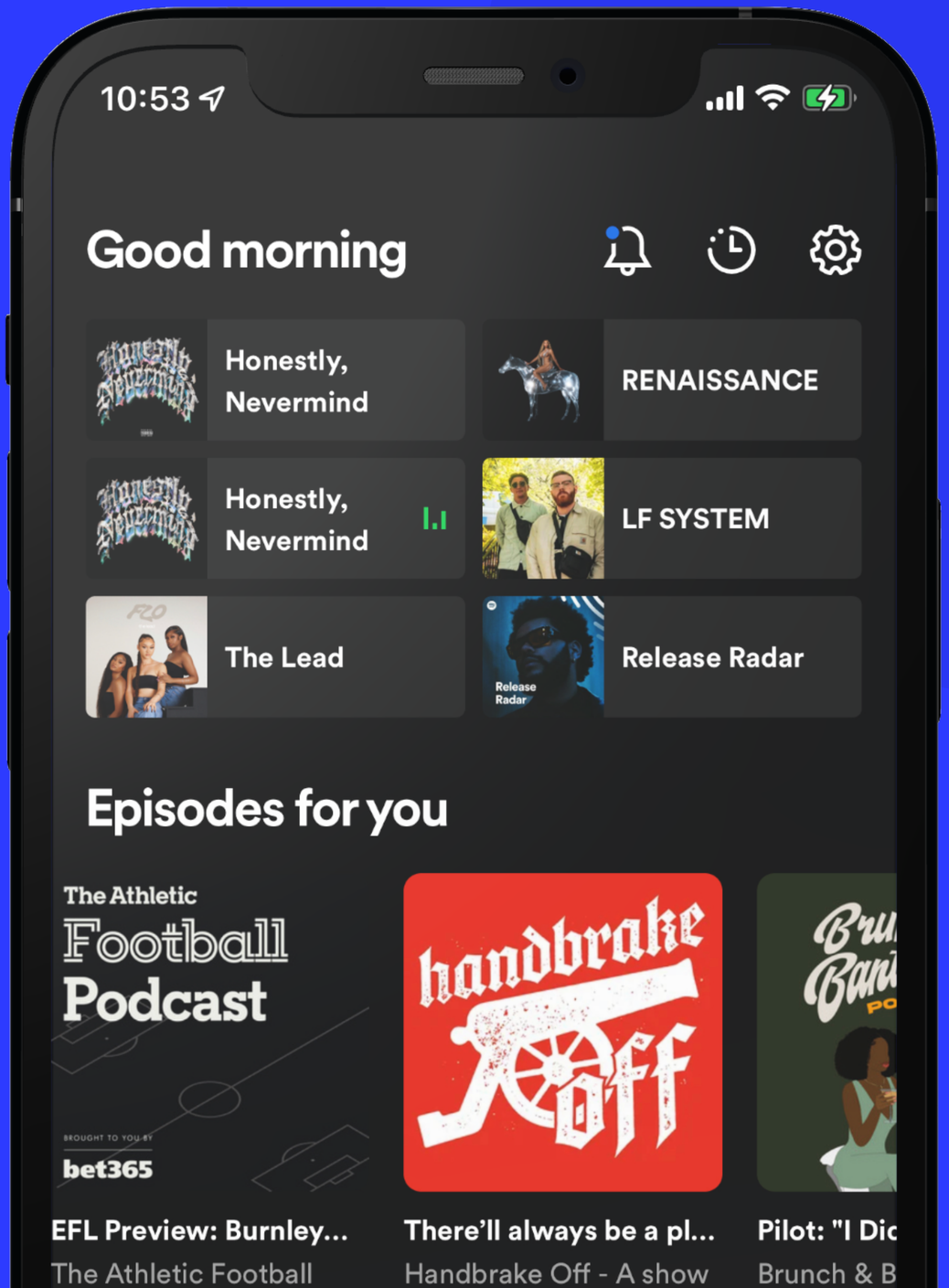
```swift
final class MusicViewModel: ObservableObject {
    @Published private(set) var songs: [Song] = []

    @MainActor
    func fetchSongs() async {
        self.songs = try? await NetworkingManager.shared.request(session: .shared,
                                                                 .songs,
                                                                 type: [SongsResponse].self) ?? []
    }
}
```

UI updates are now handled on the main thread

🤔

Multiple Resources

How can we get recommendations & podcasts?



```swift
struct HomeContentManager {

    private let mapper = ContentMapper()

    func fetchContent(completion: @escaping (Result<[ContentItem], Error>) -> Void) {

        NetworkingManager.shared.request(.recommended,
                                         type: RecommendedResponse.self) { recommenededRes in

            switch recommenededRes {

            case .success(let recommendedResponse):

                NetworkingManager.shared.request(.podcasts,
                                                 type: PodcastsResponse.self) { podcastRes in

                    switch podcastRes {

                    case .success(let podcastResponse):

                        let mappedRecommended = mapper.map(recommended: recommendedResponse)
                        let mappedPodcast = mapper.map(podcasts: podcastResponse)
                        let contentItem = [mappedRecommended, mappedPodcast]
                        completion(.success(contentItem))

                    case .failure(let error):

                        completion(.failure(error))

                    }

                }

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


```
struct HomeContentManager {

    private let mapper = ContentMapper()

    func fetchContent(completion: @escaping (Result<[ContentItem], Error>) -> Void

        NetworkingManager.shared.request(.recommended,
                                         type: RecommendedResponse.self) { recomme

            switch recommenededRes {

            case .success(let recommendedResponse):

                NetworkingManager.shared.request(.podcasts,
                                                 type: PodcastsResponse.self) { po

                    switch podcastRes {

                    case .success(let podcastResponse):

                        let mappedRecommended = mapper.map(recommended: recommende
                        let mappedPodcast = mapper.map(podcasts: podcastResponse)
                        let contentItem = [mappedRecommended, mappedPodcast]
                        completion(.success(contentItem))

                    case .failure(let error):

                        completion(.failure(error))

                    }

                }

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

```swift
struct HomeContentManager {

    private let mapper = ContentMapper()

    func fetchContent() async throws -> [ContentItem] {

        let recommended = try await NetworkingManager.shared.request(session: .shared,
                                                                     .recommended,
                                                                     type: RecommendedResponse.self)
        let podcasts = try await NetworkingManager.shared.request(session: .shared,
                                                                 .podcasts,
                                                                 type: PodcastsResponse.self)


        let mappedRecommended = mapper.map(recommended: recommended)
        let mappedPodcast = mapper.map(podcasts: podcasts)

        return [mappedRecommended, mappedPodcast]
    }
}
```

```swift
struct HomeContentManager {

    private let mapper = ContentMapper()

    func fetchContent() async throws -> [ContentItem] {

        let recommended = try await NetworkingManager.shared.request(session: .shared,
                                                                     .recommended,
                                                                     type: RecommendedResponse.self)
        let podcasts = try await NetworkingManager.shared.request(session: .shared,
                                                                 .podcasts,
                                                                 type: PodcastsResponse.self)

        let mappedRecommended = mapper.map(recommended: recommended)
        let mappedPodcast = mapper.map(podcasts: podcasts)

        return [mappedRecommended, mappedPodcast]
    }
}
```

STRUCTURED CONCURRENCY

You can see your async code in clear sequential order

AsyncLet

```
struct HomeContentManager {

    private let mapper = ContentMapper()

    func fetchContent() async throws -> [ContentItem] {

        let recommended = try await NetworkingManager.shared.request(session: .shared,
                                                                     .recommended,
                                                                     type: RecommendedResponse.self)
        let podcasts = try await NetworkingManager.shared.request(session: .shared,
                                                                 .podcasts,
                                                                 type: PodcastsResponse.self)


        let mappedRecommended = mapper.map(recommended: recommended)
        let mappedPodcast = mapper.map(podcasts: podcasts)

        return [mappedRecommended, mappedPodcast]
    }
}
```

AsyncLet

```
struct HomeContentManager {

    private let mapper = ContentMapper()

    func fetchContent() async throws -> [ContentItem] {
        let recommended = try await NetworkingManager.shared.request(session: .shared,
                                                                     .recommended,
                                                                     type: RecommendedResponse.self)
        let podcasts = try await NetworkingManager.shared.request(session: .shared,
                                                                 .podcasts,
                                                                 type: PodcastsResponse.self)

        let mappedRecommended = mapper.map(recommended: recommended)
        let mappedPodcast = mapper.map(podcasts: podcasts)

        return [mappedRecommended, mappedPodcast]
    }
}
```

```swift
struct HomeContentManager {

    private let mapper = ContentMapper()

    func fetchContent() async throws -> [ContentItem] {

        let recommended = try await NetworkingManager.shared.request(session: .shared,
                                                                     .recommended,
                                                                     type: RecommendedResponse.self)
        let podcasts = try await NetworkingManager.shared.request(session: .shared,
                                                                 .podcasts,
                                                                 type: PodcastsResponse.self)

        let mappedRecommended = mapper.map(recommended: recommended)
        let mappedPodcast = mapper.map(podcasts: podcasts)

        return [mappedRecommended, mappedPodcast]
    }
}
```

WE CAN MAKE THIS BETTER

ASYNC LET

Helps us run a defined number of async functions in parallel

AsyncLet

```
struct HomeContentManager {

    private let mapper = ContentMapper()

    func fetchContent() async throws -> [ContentItem] {

        let recommended = try await NetworkingManager.shared.request(session: .shared,
                                                                     .recommended,
                                                                     type: RecommendedResponse.self)
        let podcasts = try await NetworkingManager.shared.request(session: .shared,
                                                                 .podcasts,
                                                                 type: PodcastsResponse.self)


        let mappedRecommended = mapper.map(recommended: recommended)
        let mappedPodcast = mapper.map(podcasts: podcasts)

        return [mappedRecommended, mappedPodcast]
    }
}
```

AsyncLet

```swift
struct HomeContentManager {

    private let mapper = ContentMapper()

    func fetchContent() async throws -> [ContentItem] {

        async let recommended = NetworkingManager.shared.request(session: .shared,
                                                                 .recommended,
                                                                 type: RecommendedResponse.self)
        async let podcasts = NetworkingManager.shared.request(session: .shared,
                                                              .podcasts,
                                                              type: PodcastsResponse.self)

        let mappedRecommended = try await mapper.map(recommended: recommended)
        let mappedPodcast = try await mapper.map(podcasts: podcasts)

        return [mappedRecommended, mappedPodcast]
    }
}
```

AsyncLet

```
struct HomeContentManager {

    private let mapper = ContentMapper()

    func fetchContent() async throws -> [ContentItem] {

        async let recommended = NetworkingManager.shared.request(session: .shared,
                                                                 .recommended,
                                                                 type: RecommendedResponse.self)
        async let podcasts = NetworkingManager.shared.request(session: .shared,
                                                              .podcasts,
                                                              type: PodcastsResponse.self)

        let mappedRecommended = try await mapper.map(recommended: recommended)
        let mappedPodcast = try await mapper.map(podcasts: podcasts)

        return [mappedRecommended, mappedPodcast]
    }
}
```

AsyncLet

```
struct HomeContentManager {

    private let mapper = ContentMapper()

    func fetchContent() async throws -> [ContentItem] {

        async let recommended = NetworkingManager.shared.request(session: .shared,
                                                                 .recommended,
                                                                 type: RecommendedResponse.self)
        async let podcasts = NetworkingManager.shared.request(session: .shared,
                                                              .podcasts,
                                                              type: PodcastsResponse.self)

        let mappedRecommended = try await mapper.map(recommended: recommended)
        let mappedPodcast = try await mapper.map(podcasts: podcasts)

        return [mappedRecommended, mappedPodcast]
    }
}
```

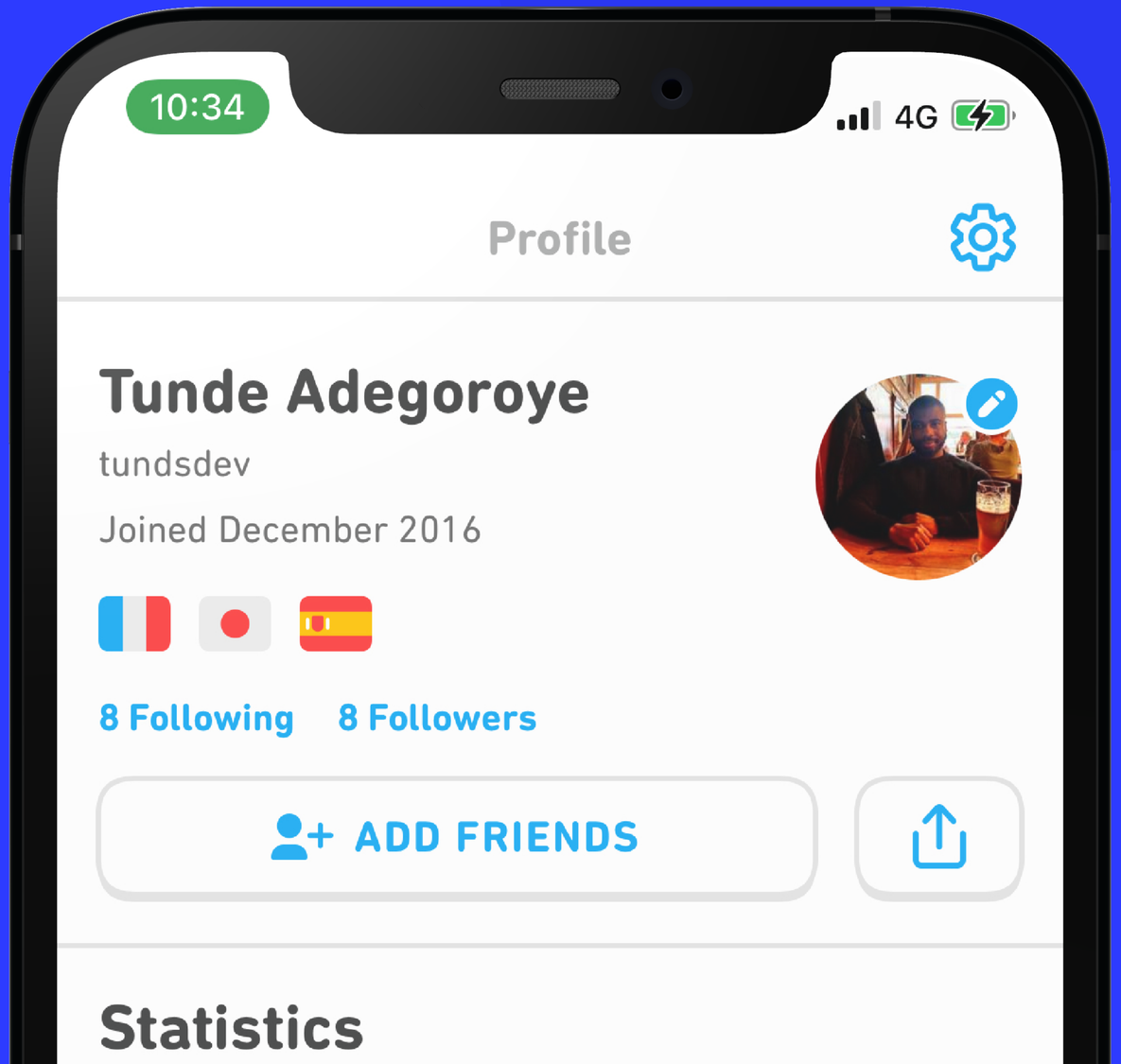
AsyncLet

```
struct HomeContentManager {

    private let mapper = ContentMapper()

    func fetchContent() async throws -> [ContentItem] {

        async let recommended = NetworkingManager.shared.request(session: .shared,
                                                                 .recommended,
                                                                 type: RecommendedResponse.self)
        async let podcasts = NetworkingManager.shared.request(session: .shared,
                                                              .podcasts,
                                                              type: PodcastsResponse.self)

        let mappedRecommended = try await mapper.map(recommended: recommended)
        let mappedPodcast = try await mapper.map(podcasts: podcasts)

        return [mappedRecommended, mappedPodcast]
    }
}
```

TaskGroup

Fetching Images

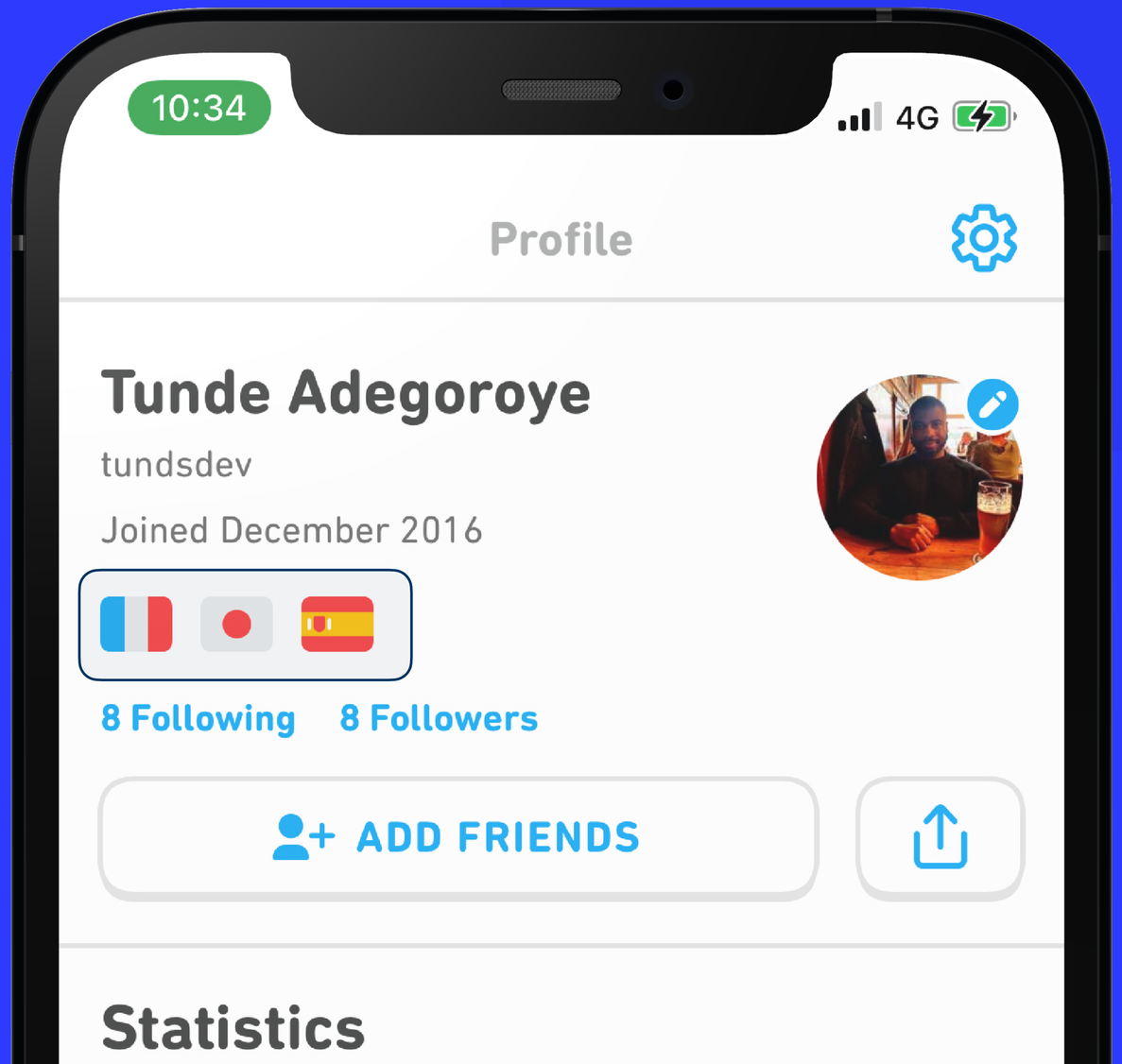
How can we get an undefined list of images?



TaskGroup

Fetching Images

How can we get an undefined list of images?



WHAT IS A TASKGROUP?

Allow us to execute an undefined collection of async tasks concurrently, waiting for all of the child tasks to finish

👬

TaskGroup

🏘️

Collection

Holds a collection of child tasks

💅🏾

Independent

They run in parallel

🏎️

No order

No guarantee on which child task will finish first

🚦

Be Careful

⚠️ A group finishes when all child tasks finish, if you're loading a lot of data this might not be the one for you

👀

Returns or Throws

Task groups can return a val, void or throw an error

TaskGroup

```swift
func fetchPhotos() async throws -> [Picture] {

    var photos = [Picture]()
    let fetchedPhotos = try await retrievePhotosList()
    try await withThrowingTaskGroup(of: Picture.self) { group in
        for photo in fetchedPhotos {
            group.addTask {
                let picture = Picture(image: try await fetchPhotoThumbnail(from: photo),
                                      author: photo.author)
                return picture
            }
        }

        for try await item in group {
            photos.append(item)
        }
    }

    return photos
}
```

TaskGroup

```
func fetchPhotos() async throws -> [Picture] {
}
```

Returns all pictures

Asynchronously will get us back an array of pictures we can use in our app

TaskGroup

```swift
func fetchPhotos() async throws -> [Picture] {

    var photos = [Picture]()
    let fetchedPhotos = try await retrievePhotosList()

}
```

Async get pics

Asynchronously get a collection of our urls that we will load later

TaskGroup

```swift
func fetchPhotos() async throws -> [Picture] {

    var photos = [Picture]()
    let fetchedPhotos = try await retrievePhotosList()

    try await withThrowingTaskGroup(of: Picture.self) { group in

    }
}
```

Create our group

Define a throwing task group that will hold our child tasks

TaskGroup

```swift
func fetchPhotos() async throws -> [Picture] {

    var photos = [Picture]()
    let fetchedPhotos = try await retrievePhotosList()

    try await withThrowingTaskGroup(of: Picture.self) { group in

    }
}
```

Child Task Return Type

Define our child tasks will return a specific type

TaskGroup

```swift
func fetchPhotos() async throws -> [Picture] {

    var photos = [Picture]()
    let fetchedPhotos = try await retrievePhotosList()
    try await withThrowingTaskGroup(of: Picture.self) { group in

        for photo in fetchedPhotos {
            group.addTask {
                let picture = Picture(image: try await fetchPhotoThumbnail(from: photo),
                                      author: photo.author)
                return picture
            }
        }

    }
}
```

Create child tasks

Define the async logic you want to execute in your child and add it to the group

👬

TaskGroup

```
func fetchPhotos() async throws -> [Picture] {

    var photos = [Picture]()
    let fetchedPhotos = try await retrievePhotosList()
    try await withThrowingTaskGroup(of: Picture.self) { group in
        for photo in fetchedPhotos {
            group.addTask {
                let picture = Picture(image: try await fetchPhotoThumbnail(from: photo),
                                      author: photo.author)
                return picture
            }
        }

        for try await item in group {
            photos.append(item)
        }
    }
}
```

Access Child

Get all the pictures asynchronously and append it to our array

TaskGroup

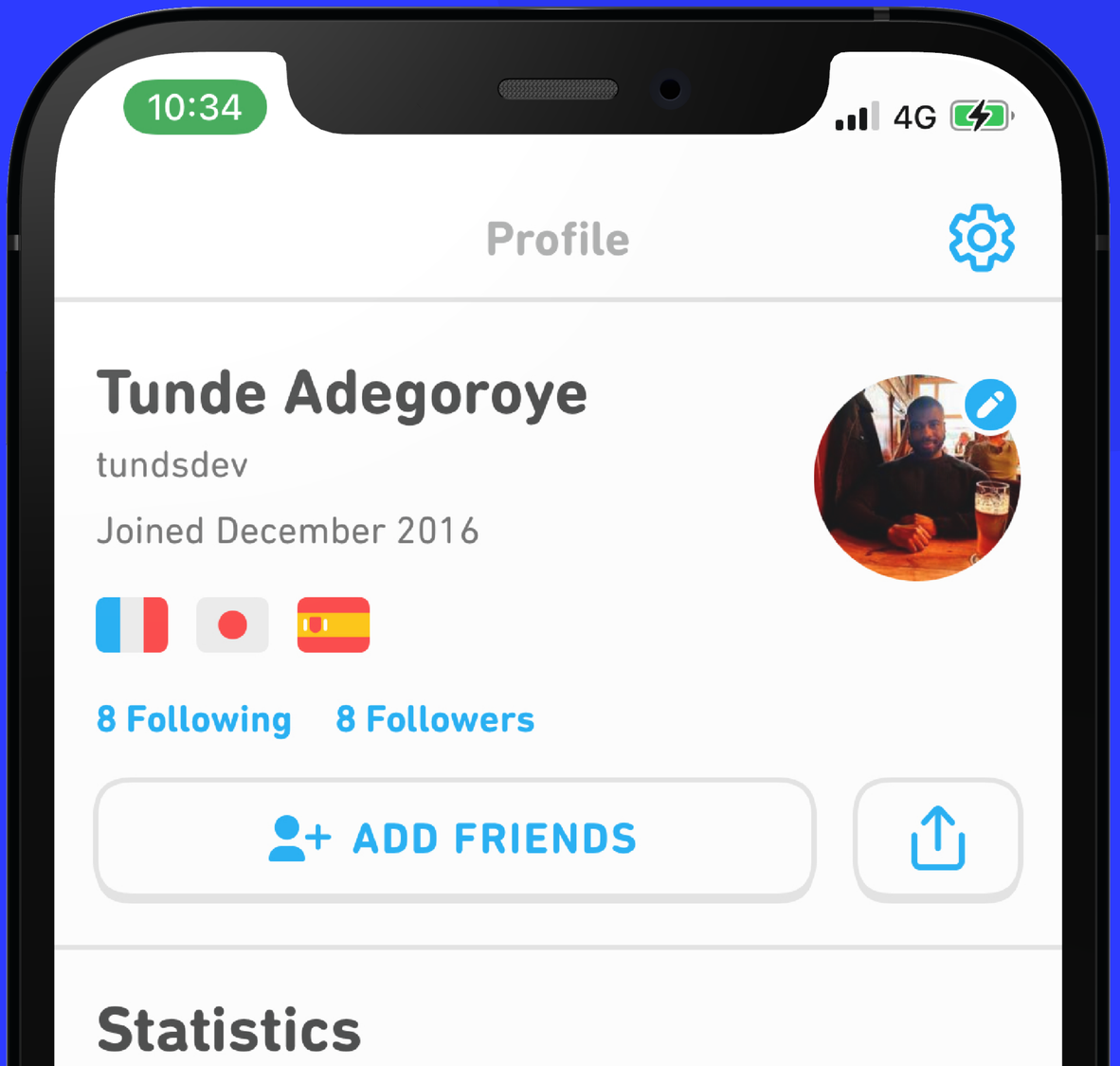
```swift
func fetchPhotos() async throws -> [Picture] {

    var photos = [Picture]()
    let fetchedPhotos = try await retrievePhotosList()
    try await withThrowingTaskGroup(of: Picture.self) { group in
        for photo in fetchedPhotos {
            group.addTask {
                let picture = Picture(image: try await fetchPhotoThumbnail(from: photo),
                                      author: photo.author)
                return picture
            }
        }

        for try await item in group {
            photos.append(item)
        }
    }

    return photos
}
```

Give back the pictures

return the collection

Caching Images

Where can we store our images?



Actors

```swift
class ArtworkCache {

    static let shared = ArtworkCache()

    private init() { }

    private lazy var cache: NSCache<NSString, UIImage> = {
        let cache = NSCache<NSString, UIImage>()
        cache.countLimit = 100
        cache.totalCostLimit = 50 * 1024 * 1024 // 52428800 Bytes > 50MB
        return cache
    }()

    func set(object: UIImage, for key: NSString) {
        cache.setObject(object, forKey: key)
    }

    func getValue(for key: NSString) -> UIImage? {
        cache.object(forKey: key)
    }
}
```

Actors

```swift
class ArtworkCache {

    static let shared = ArtworkCache()

    private init() { }

    private lazy var cache: NSCache<NSString, UIImage> = {
        let cache = NSCache<NSString, UIImage>()
        cache.countLimit = 100
        cache.totalCostLimit = 50 * 1024 * 1024 // 52428800 Bytes > 50MB
        return cache
    }()

    func set(object: UIImage, for key: NSString) {
        cache.setObject(object, forKey: key)
    }

    func getValue(for key: NSString) -> UIImage? {
        cache.object(forKey: key)
    }
}
```

Has it finished?

We don’t know when child tasks finish

Make it thread safe

How can we have a thread safe way to write to our cache

Stop data races

How can we prevent data races from occurring

ACTORS

Actors allow you to protect your state from data races

Actors

```swift
class ArtworkCache {

    static let shared = ArtworkCache()

    private init() { }

    private lazy var cache: NSCache<NSString, UIImage> = {
        let cache = NSCache<NSString, UIImage>()
        cache.countLimit = 100
        cache.totalCostLimit = 50 * 1024 * 1024 // 52428800 Bytes > 50MB
        return cache
    }()

    func set(object: UIImage, for key: NSString) {
        cache.setObject(object, forKey: key)
    }

    func getValue(for key: NSString) -> UIImage? {
        cache.object(forKey: key)
    }
}
```

Actors

```swift
actor ArtworkCache {

    static let shared = ArtworkCache()

    private init() { }

    private lazy var cache: NSCache<NSString, UIImage> = {
        let cache = NSCache<NSString, UIImage>()
        cache.countLimit = 100
        cache.totalCostLimit = 50 * 1024 * 1024 // 52428800 Bytes > 50MB
        return cache
    }()

    func set(object: UIImage, for key: NSString) {
        cache.setObject(object, forKey: key)
    }

    func getValue(for key: NSString) -> UIImage? {
        cache.object(forKey: key)
    }
}
```

Actor Keyword

Marks the object as an actor

Actors

```swift
func fetchPhotos() async throws -> [Picture] {

    var photos = [Picture]()
    let fetchedPhotos = try await retrievePhotosList()
    try await withThrowingTaskGroup(of: Picture.self) { group in
        for photo in fetchedPhotos {
            group.addTask {
                let picture = Picture(image: try await fetchPhotoThumbnail(from: photo),
                                      author: photo.author)
                return picture
            }
        }

        for try await item in group {
            photos.append(item)
        }
    }

    return photos
}
```

Actors

```swift
func fetchPhotos(cache: ArtworkCache = .shared) async throws -> [Picture] {

    var photos = [Picture]()
    let fetchedPhotos = try await retrievePhotosList()
    try await withThrowingTaskGroup(of: Picture.self) { group in
        for photo in fetchedPhotos {
            group.addTask {
                let picture = Picture(image: try await fetchPhotoThumbnail(from: photo),
                                      author: photo.author)
                return picture
            }
        }

        for try await item in group {

            await cache.set(object: item.image, for: item.author as NSString)
            photos.append(item)
        }
    }

    return photos
}
```

Inject it

Pass our actor into the function

Actors

```swift
func fetchPhotos(cache: ArtworkCache = .shared) async throws -> [Picture] {

    var photos = [Picture]()
    let fetchedPhotos = try await retrievePhotosList()
    try await withThrowingTaskGroup(of: Picture.self) { group in
        for photo in fetchedPhotos {
            group.addTask {
                let picture = Picture(image: try await fetchPhotoThumbnail(from: photo),
                                      author: photo.author)
                return picture
            }
        }

        for try await item in group {

            await cache.set(object: item.image, for: item.author as NSString)
            photos.append(item)
        }
    }

    return photos
}
```

Wait

Safely update the actor

WHAT ARE TASKS?

Allow you to execute asynchronous code

🔨

Tasks

👀

Priority

Control the priority level of tasks

💑

Control Relationships

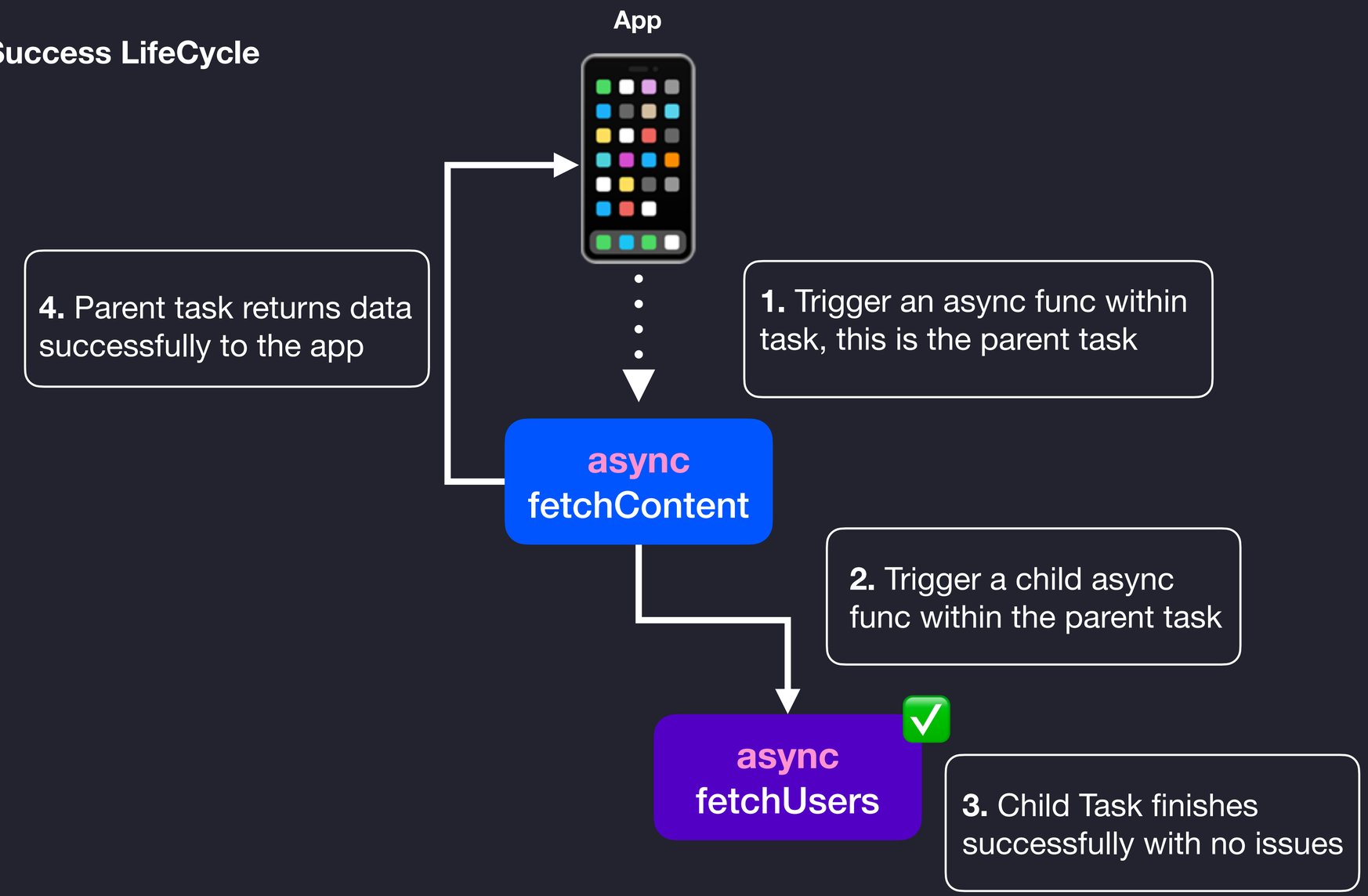
How do they relate to one another

❌

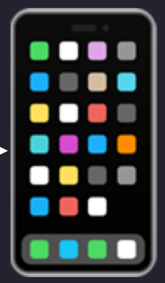
Handle Cancellation

Do you want to cancel tasks?

Task Success LifeCycle



Task Failure LifeCycle



1. Trigger an async func within task, this is the parent task

2. Trigger two child tasks func within the parent task

3. fetchUsers fails and throws an error

4. fetchPosts func is cancelled with cancellation error

5. Cancellation will trickle down

6. Cancellation Error is thrown from parent task to the app

async fetchContent

async fetchPosts(id:)

async fetchUsers

async cachePosts

Tasks

```swift
// MARK - SwiftUI Task Modifier Example
// iOS15+

MyView()
   .task {
      await fetchData()
}
```

Tasks

```swift
// MARK - SwiftUI Task Modifier Example
// iOS15+

MyView()
   .task {
      await fetchData()
}
// MARK - SwiftUI Task Modifier Example

func executeRequest() async {
    Task {
        await fetchData()
    }
}
```

Tasks

```swift
// MARK - SwiftUI Task Modifier Example

func executeRequest() async {
    Task {
        await fetchData()
    }
```

MANAGE YOUR CANCELLATION

Going Backwards

- Fixed an issue where a project using a CocoaPods installation without a “Debug” configuration set up could have a mismatch between building and semantic functionality in the editor, leading to problems like the editor showing “live issues” that don’t exist when building. (84502615) (FB9717206)

StoreKit

New Features

- StoreKit testing includes new subscription renewal rates in terms of months. If you test on operating systems earlier than macOS Monterey 12.1, iOS 15.2, tvOS 15.2, and watchOS 8.3, the operating system uses the deprecated rates in terms of days to approximate the renewal rates. (76929977)

Swift

New Features

- You can now use Swift Concurrency in applications that deploy to macOS Catalina 10.15, iOS 13, tvOS 13, and watchOS 6 or newer. This support includes `async`/`await`, actors, global actors, structured concurrency, and the task APIs. (70738378)

Resolved Issues

- Fixed an issue that prevented Swift libraries depending on Combine from building for targets including armv7 and i386 architectures. (82183186, 82189214)
- Fixed an issue that caused availability checks in iPhone and iPad apps running on a Mac with Apple silicon to always return `true`, which caused iOS apps running in macOS Big Sur to see iOS 15 APIs as available, resulting in crashes. These availability checks now return the correct result for apps compiled with Xcode 13.2. (83378814)
- Mac apps built with Mac Catalyst that use Swift Concurrency now launch in an operating system prior to macOS 12 Monterey. (84393581)
- watchOS apps that use Swift Concurrency and deploy prior to watchOS 8 now build for 64-bit watchOS simulator

Going Backwards

```swift
@available(swift, introduced: 5.5, message: "Use async alternative of retrieveQuotes")
func retrieveQuotes(completion: @escaping (Result<[Quote], Error>) -> Void) { ••• }

func retrieveQuotes() async throws -> [Quote] {

    try await withCheckedThrowingContinuation { continuation in

        service.fetchQuotes { res in

            do {

                continuation.resume(returning: try res.get())

            } catch {

                continuation.resume(throwing: error)

            }
        }
    }
}
```

Going Backwards

```swift
@available(swift, introduced: 5.5, message: "Use async alternative of retrieveQuotes")
func retrieveQuotes(completion: @escaping (Result<[Quote], Error>) -> Void) { ••• }

func retrieveQuotes() async throws -> [Quote] {

    try await withCheckedThrowingContinuation { continuation in

        service.fetchQuotes { res in

            do {

                continuation.resume(returning: try res.get())

            } catch {

                continuation.resume(throwing: error)

            }
        }
    }
}
```

Going Backwards

```swift
@available(swift, introduced: 5.5, message: "Use async alternative of retrieveQuotes")
func retrieveQuotes(completion: @escaping (Result<[Quote], Error>) -> Void) { ••• }

func retrieveQuotes() async throws -> [Quote] {

    try await withCheckedThrowingContinuation { continuation in

        service.fetchQuotes { res in

            do {

                continuation.resume(returning: try res.get())

            } catch {

                continuation.resume(throwing: error)

            }
        }
    }
}
```

Going Backwards

```swift
@available(swift, introduced: 5.5, message: "Use async alternative of retrieveQuotes")
func retrieveQuotes(completion: @escaping (Result<[Quote], Error>) -> Void) { ••• }

func retrieveQuotes() async throws -> [Quote] {

    try await withCheckedThrowingContinuation { continuation in

        service.fetchQuotes { res in

            do {

                continuation.resume(returning: try res.get())

            } catch {

                continuation.resume(throwing: error)

            }
        }
    }
}
```

Summary

Async/Await

Async/Await allows to write & manage async code easily

Main Actor

Gives us a way to place UI updates back onto the main thread

AsyncLet

Allows us to run async func that aren't dependent on one another in parallel

TaskGroups

Allow us to execute an undefined collection of async tasks in parallel

Actors

Give us a safe way to mutate state across multiple threads

Tasks

Allow us to execute async code

Continuations

Give us a nice way to gracefully migrate our legacy code to use Swift Concurrency

AND THERE'S MORE...I DON'T HAVE TIME FOR LOL

FREE Swift Concurrency Course

On my channel called **Learn Swift Concurrency Online For Free**

https://youtu.be/U6lQustiTGE

Swift Async Algorithms

github.com/apple/swift-async-algorithms

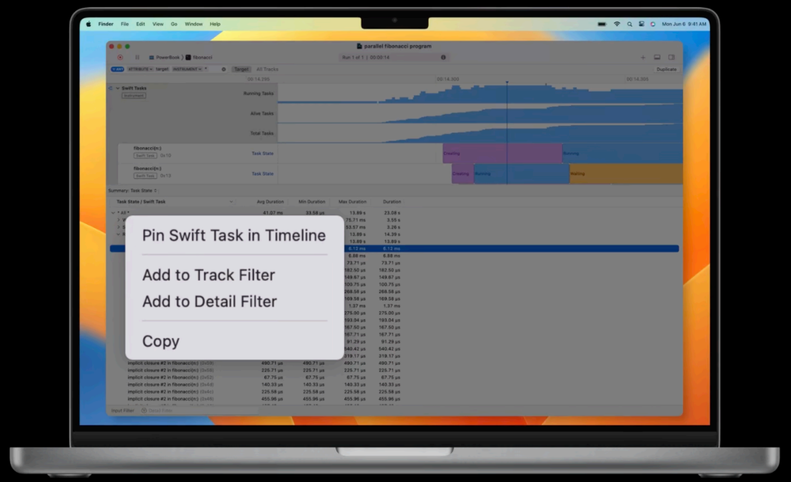
Meet Swift Async Algorithms

Discover the latest open source Swift package from Apple: Swift Async Algorithms. We'll explore algorithms from this package that you can use with AsyncSequence, including zip, merge, and throttle. Follow along with us as we use these algorithms to build a great messaging app. We'll also share best practices for combining multiple AsyncSequences and using the Swift Clock type to work with values over time. To get the most out of this session, we recommend watching "Meet AsyncSequence."

Resources

‹ More Videos

Using the Swift Concurrency Template



Overview Transcript

Visualize and optimize Swift concurrency

Learn how you can optimize your app with the Swift Concurrency template in Instruments. We'll discuss common performance issues and show you how to use Instruments to find and resolve these problems. Learn how you can keep your UI responsive, maximize parallel performance, and analyze Swift concurrency activity within your app. To get the most out of this session, we recommend familiarity with Swift concurrency (including tasks and actors).

Resources

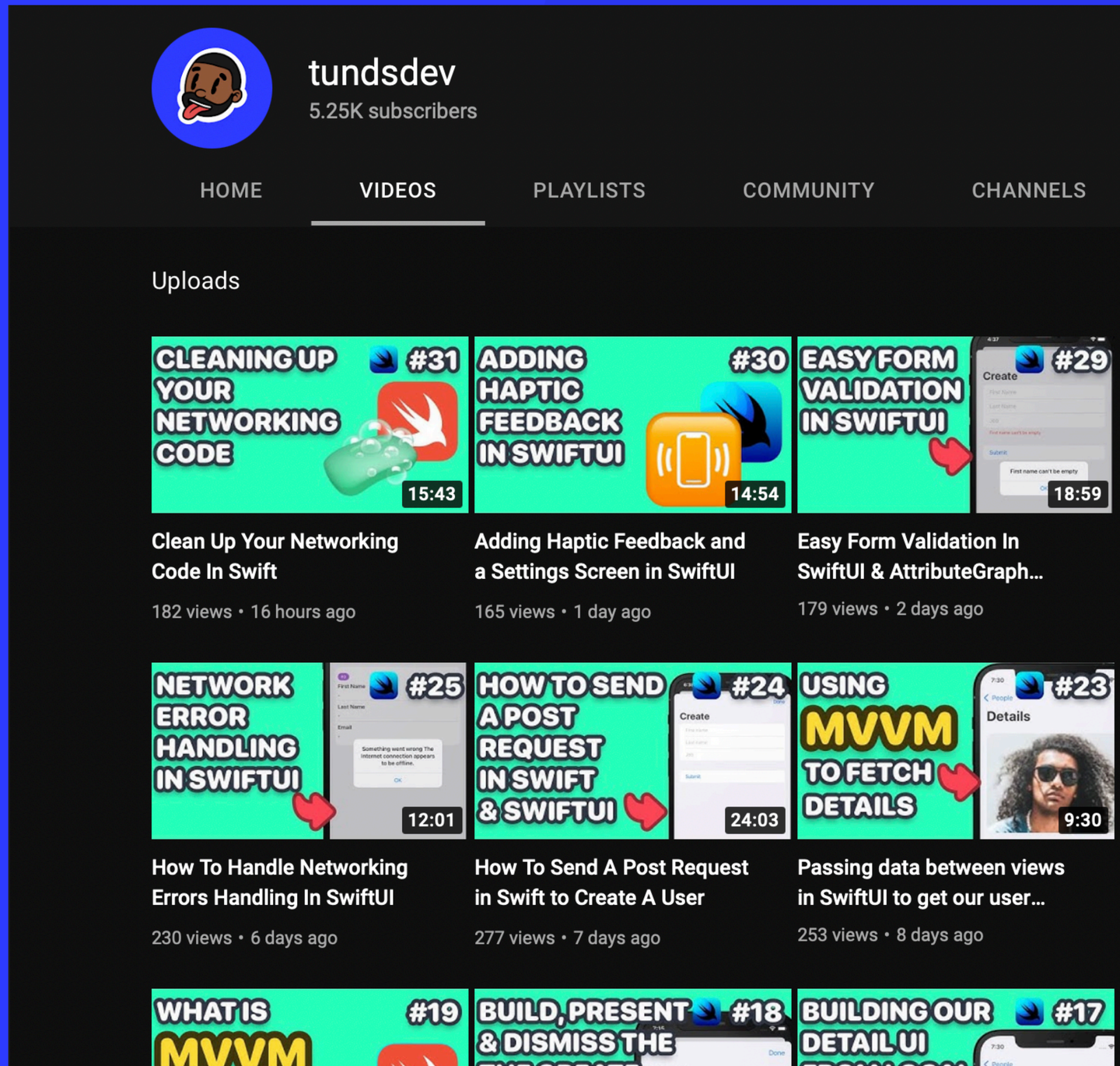
I'D USE SWIFT CONCURRENCY

Subscribe to my channel

Search for **tundsdev** on YouTube

SUBSCRIBE

https://www.youtube.com/c/tundsdev



👀 Find more free courses on my channel

Look for the playlists on **tundsdev**

💕 SwiftUI Sessions

Learn everything, you need to know
to get started with SwiftUI

⚡ SwiftUI Take Home Test

Learn everything, you need to know
to ace your next take-home test

🏄‍♂️ SwiftUI Data Flow

Understand how state & data flow
works in SwiftUI

📱 SwiftUI MVVM Examples

Learn how to structure your apps & projects
using MVVM. As well as implementing
networking

Come & Grab A Laptop Sticker

If you'd like a free sticker come and
grab me there's only a limited amount

Come & Grab A Laptop Sticker

If you'd like a free sticker come and grab me there's only a limited amount

SUBSCRIBE

Subscribe to my channel

Search for **tundsdev** on YouTube

Follow me on Twitter

@tundsdev

THANK YOU & DEUCES ✌🏾