Les requêtes réseaux

Antoine Moevus

Table des matières

References

1. https://matteomanferdini.com/swift-singleton/

Singleton

Le Singleton est un patron de conception (Design Pattern) qui garantit qu'une classe n'a qu'une seule instance, et fournit un point d'accès global à celle-ci. Ce modèle est utilisé lorsque vous voulez vous assurer qu'il y a une et une seule instance d'une classe particulière qui est partagée et accessible à partir de n'importe quel endroit de votre application.

Utilité

Le modèle Singleton est utile lorsque vous voulez que des parties distinctes de votre code soient capables d'accéder facilement à une instance partagée d'une classe particulière. Par exemple, vous pourriez avoir besoin d'un gestionnaire de réseau unique pour coordonner toutes les requêtes réseau ou d'un gestionnaire de base de données unique pour gérer toutes les opérations de base de données.

Fonctionnement

En Swift, le Singleton est généralement implémenté en définissant une propriété, arbitrairement nommée `shared` (ou parfois `default`), statique dans une classe, et en rendant son initialisateur privé pour s'assurer qu'il ne peut pas être appelé directement.

```
class Singleton {
    static let shared = Singleton()

    private init() {
        // Private initialization to ensure just one instance is created.
    }
}
```

Ainsi, vous pouvez accéder à l'instance Singleton partout dans votre code en utilisant `Singleton.shared`.

Exemples concrets

Un exemple célèbre d'utilisation du modèle Singleton dans iOS est la classe `UIApplication`. Vous pouvez accéder à l'instance d'application partagée n'importe où dans votre code en utilisant `UIApplication.shared`. Un autre exemple est la classe `URLSession` pour les opérations de réseau, où `URLSession.shared` est utilisé pour effectuer des tâches de réseau basées sur des sessions de réseau partagées.

Cependant, l'utilisation du Singleton doit être faite avec prudence, car elle peut rendre le code difficile à tester et à maintenir en raison de ses dépendances globales et de son état partagé. De plus, une mauvaise utilisation du Singleton peut conduire à des problèmes de concurrence et de sécurité des données.

Exemple pour une classe `Jeu`

Dans le contexte d'un jeu, un singleton pourrait être utilisé pour gérer l'état global du jeu, comme les scores, les paramètres, et d'autres données partagées. Voici un exemple d'implémentation :

```
class Jeu {
    static let shared = Jeu()

    var score: Int = 0
    var level: Int = 1
    var highScore: Int = 0
    var settings: [String: Any] = [:]

    private init() {}
}

// Utilisation :
Jeu.shared.score = 100
Jeu.shared.level = 2
Jeu.shared.settings["soundEnabled"] = true
```

Dans cet exemple, nous avons une classe `Jeu` qui gère l'état global du jeu. Elle maintient des informations comme le score actuel, le niveau actuel, le score le plus élevé et divers paramètres du jeu. Toutes ces informations sont accessibles de n'importe quel endroit de votre application via l'instance partagée `Jeu.shared`.

Mais comme toujours, faites attention à l'utilisation des singletons. Assurez-vous que votre code reste maintenable et facile à tester, et évitez de trop dépendre des états globaux partagés.

Grand Central Dispatch (GCD)

Grand Central Dispatch (GCD) est une bibliothèque de bas niveau que Apple fournit pour gérer le travail concurrent ou parallèle dans votre application. Il offre une façon efficace de gérer l'asynchronisme et le multithreading dans votre application.

Qu'est-ce que le GCD?

GCD est une technologie développée par Apple qui permet d'optimiser l'application en tirant parti de tous les cœurs du processeur disponibles dans l'appareil. Elle permet de distribuer les tâches sur différentes files d'exécution, ce qui peut améliorer les performances en utilisant les ressources du système plus efficacement.

Comment utiliser le GCD?

Pour utiliser le GCD, vous enverrez des unités de travail (c'est-à-dire des blocs de code ou des fermetures) à différentes files d'attente. Ces files d'attente gèrent tout le travail de planification pour vous. Vous pouvez choisir entre les files d'attente sérialisées et concurrentes.

Voici un exemple simple :

```
DispatchQueue.global(qos: .background).async {
    print("Ceci est exécuté en arrière-plan")
    DispatchQueue.main.async {
        print("Ceci est exécuté sur le thread principal")
    }
}
```

Dans cet exemple, un bloc de travail est envoyé à une file d'attente globale avec un niveau de qualité de service (QoS) d'arrière-plan. Une fois ce travail terminé, un autre bloc de travail est envoyé à la file d'attente principale pour mettre à jour l'interface utilisateur.

Pourquoi utiliser le GCD?

GCD offre plusieurs avantages :

- Il vous permet d'écrire du code non bloquant pour améliorer la réactivité de votre application.
- Il vous permet de tirer parti du multithreading sans avoir à gérer directement les threads.
- Il vous permet de donner des priorités à différentes tâches, de sorte que les tâches importantes puissent être exécutées en premier.
- Il vous offre la possibilité de créer du code plus propre et plus lisible en encapsulant des tâches liées dans des blocs de code ou des fermetures.

GCD est une technologie puissante qui peut aider à améliorer les performances de votre application. Cependant, comme avec toute technologie, il est important de comprendre comment et

quand l'utiliser correctement pour en tirer le meilleur parti.

Files d'attente

Une file d'attente (queue) est une structure de données qui gère les blocs de travail (work items) qui doivent être exécutés. Chaque file d'attente a une politique particulière pour déterminer l'ordre dans lequel les blocs de travail sont exécutés.

Il existe deux types principaux de files d'attente :

- Les files d'attente sérialisées : Elles exécutent un bloc de travail à la fois, dans l'ordre où ils ont été ajoutés à la file d'attente. Les nouveaux blocs de travail attendent que le bloc de travail précédent soit terminé avant de commencer.
- Les files d'attente concurrentes : Elles peuvent exécuter plusieurs blocs de travail à la fois. Les blocs de travail commencent dans l'ordre dans lequel ils ont été ajoutés à la file d'attente, mais ils peuvent terminer dans n'importe quel ordre, en fonction de combien de temps il faut pour exécuter chaque bloc de travail.

De plus, Swift fournit deux files d'attente spécifiques pour des utilisations particulières :

- La file d'attente principale : C'est une file d'attente sérialisée qui exécute des travaux sur le thread principal. Elle est principalement utilisée pour mettre à jour l'interface utilisateur, car toutes les mises à jour de l'interface utilisateur doivent être effectuées sur le thread principal.
- Les files d'attente globales : Ce sont des files d'attente concurrentes que Swift fournit pour exécuter des travaux en arrière-plan. Elles ont différents niveaux de qualité de service (Quality of Service - QoS) pour indiquer l'importance ou la priorité des travaux qui sont ajoutés à la file d'attente.

Exemples de files d'attente

```
// File d'attente principale
DispatchQueue.main.async {
    // Mettre à jour l'interface utilisateur
    self.myLabel.text = "Mise à jour de l'interface utilisateur sur la file d'attente
principale"
}

// File d'attente globale avec QoS par défaut
DispatchQueue.global().async {
    // Exécuter un travail en arrière-plan
    let data = downloadDataFromServer()
    DispatchQueue.main.async {
        // Une fois les données téléchargées, mettre à jour l'interface utilisateur
sur la file d'attente principale
        self.myLabel.text = data
    }
}
```

```
// File d'attente globale avec un QoS spécifique
DispatchQueue.global(qos: .userInitiated).async {
    // Exécuter un travail en arrière-plan qui a été initié par l'utilisateur
    let data = downloadDataFromServer()
    DispatchQueue.main.async {
        // Une fois les données téléchargées, mettre à jour l'interface utilisateur
sur la file d'attente principale
        self.myLabel.text = data
    }
}
```

Dans cet exemple, `DispatchQueue.global(qos: .userInitiated).async` est utilisé pour effectuer le téléchargement des données. C'est un travail qui peut prendre un certain temps, et vous ne voulez pas bloquer la file d'attente principale (qui gère l'interface utilisateur) pendant qu'il est en cours d'exécution. C'est pourquoi vous l'exécutez sur une file d'attente globale en arrière-plan.

Cependant, une fois que les données sont téléchargées, vous voulez mettre à jour l'interface utilisateur pour afficher ces données. Toutes les mises à jour de l'interface utilisateur doivent être effectuées sur la file d'attente principale. Si vous essayez de faire une mise à jour de l'interface utilisateur à partir d'une file d'attente en arrière-plan, vous pouvez obtenir des comportements imprévisibles. Par conséquent, une fois que les données sont téléchargées, vous enfilez une tâche sur la file d'attente principale pour mettre à jour l'interface utilisateur. C'est ce que fait `DispatchQueue.main.async`.

La sous-sous-section suivant décrit l'argument `qos`.

Qualité de Service (QoS)

La qualité de service (QoS) est un mécanisme utilisé par Grand Central Dispatch (GCD) pour différencier les tâches en fonction de leur importance. Les QoS permettent au système d'optimiser l'utilisation du CPU, de l'énergie et d'autres ressources système en équilibrant les tâches en fonction de leur priorité.

Les niveaux de qualité de service, de la plus haute priorité à la plus faible, sont les suivants :

- `userInteractive`
- `userInitiated`
- `default`
- `utility`
- `background`

Chaque niveau de QoS a des caractéristiques spécifiques et est utilisé dans différents contextes.

```
// QoS : userInteractive
DispatchQueue.global(qos: .userInteractive).async {
    // Tâches à haute priorité qui mettent à jour l'interface utilisateur. Ces tâches
sont généralement de courte durée et doivent être exécutées immédiatement avant que la
```

```
vue soit mise à jour pour fournir une expérience utilisateur réactive.
}

// QoS : userInitiated
DispatchQueue.global(qos: .userInitiated).async {
    // Tâches à haute priorité initiées par l'utilisateur qui nécessitent des résultats immédiats. Ces tâches préviennent l'utilisateur qu'une activité est en cours et incluent des opérations comme charger des données pour une nouvelle vue.
}

// QoS : default
DispatchQueue.global(qos: .default).async {
    // Tâches avec la priorité par défaut si aucun autre niveau n'est spécifié. Utilisez ce niveau pour les tâches qui ne nécessitent pas une exécution immédiate et n'ont pas d'effet visible pour l'utilisateur.
}

// QoS : utility
DispatchQueue.global(qos: .utility).async {
    // Tâches à faible priorité pour des tâches de longue durée que l'utilisateur est conscient qu'elles prennent du temps. Par exemple, les tâches de maintenance de l'application ou les téléchargements de contenu en arrière-plan.
}

// QoS : background
DispatchQueue.global(qos: .background).async {
    // Tâches à la plus faible priorité pour des tâches qui ne nécessitent pas l'interaction de l'utilisateur et peuvent s'exécuter en arrière-plan pendant que l'application est inactive. Par exemple, les sauvegardes de base de données, la mise à jour des données pour une utilisation ultérieure, le nettoyage et l'archivage des données.
}
```

URLSession

URLSession est une classe du cadriciel Foundation qui fournit une interface pour effectuer des tâches réseau, telles que l'obtention de données à partir d'un serveur ou le téléchargement et l'upload de fichiers. Elle gère également des fonctionnalités avancées, comme la gestion des cookies, l'authentification HTTPS, la redirection, et bien d'autres.

Création de session

Pour créer une session, on utilise la fonction `URLSession.shared`, qui retourne une session singleton partagée par tout le processus :

```
let session = URLSession.shared
```

Création de tâches

La classe URLSession propose trois types de tâches :

- URLSessionDataTask : Utilisé pour envoyer et recevoir des données vers et depuis un serveur.
- URLSessionUploadTask : Utilisé pour uploader des fichiers vers un serveur.
- URLSessionDownloadTask : Utilisé pour télécharger des fichiers à partir d'un serveur.

Par exemple, pour créer une URLSessionDataTask, on fait :

```
let url = URL(string: "https://www.example.com")!
let task = URLSession.shared.dataTask(with: url) { (data, response, error) in
    if let error = error {
        print("Erreur : \(error)")
    } else if let data = data {
        // Faire quelque chose avec les données
    }
}
task.resume() // Ne pas oublier d'appeler resume() pour démarrer la tâche
```

Gestion des erreurs

Les erreurs de tâche de session sont signalées par le paramètre `error` du handler de complétion. Vous pouvez vérifier ce paramètre pour vous assurer que la tâche a réussi. Les erreurs peuvent inclure des problèmes de réseau, des erreurs de serveur, des problèmes d'authentification, etc.

```
let task = URLSession.shared.dataTask(with: url) { (data, response, error) in
    if let error = error {
        print("Erreur : \(error)")
    } else if let data = data {
```

```
        // Faire quelque chose avec les données
    }
}
```

Configuration de session

URLSession prend en charge plusieurs configurations de session, ce qui vous permet de contrôler le comportement de la session et de la tâche. Par exemple, vous pouvez configurer des politiques de mise en cache, utiliser des cookies personnalisés, définir des délais, etc.

```
let configuration = URLSessionConfiguration.default
configuration.timeoutIntervalForRequest = 30.0
let session = URLSession(configuration: configuration)
```

Exemples d'application pour les requêtes réseaux

ImageDownloader

```
class ImageDownloader {
    // Singleton instance
    static let shared = ImageDownloader()

    private let queue = DispatchQueue(label: "imageDownloader", attributes:
.concurrent)

    private init() {}

    func downloadImage(_ url: URL, completion: @escaping (UIImage?) -> Void) {
        queue.async {
            let data = try? Data(contentsOf: url)
            let image = data.flatMap(UIImage.init)
            DispatchQueue.main.async {
                completion(image)
            }
        }
    }
}

// Utilisation :
let URL = "https://raw.githubusercontent.com/prof-
moevus/ressources/main/rubberDuckDebugging.png"
ImageDownloader.shared.downloadImage(url) { image in
    // Faire quelque chose avec l'image.
}
```

Dans cet exemple, `ImageDownloader` est un singleton qui possède une file d'attente GCD privée. La méthode `downloadImage(_:completion:)` utilise cette file d'attente pour télécharger des images en arrière-plan. Une fois l'image téléchargée, elle utilise la file d'attente principale pour exécuter le bloc de fin d'exécution, ce qui garantit que l'interface utilisateur est mise à jour sur le thread principal.

La requête réseau se fait via la classe `Data`, ce qui peut parfois être limitant en terme de configuration. Voici le code avec `URLSession`:

```
import UIKit

class ImageDownloader {
    // Singleton instance
    static let shared = ImageDownloader()
```

```
    private let queue = DispatchQueue(label: "imageDownloader", attributes:
.concurrent)

    private init() {}

    func downloadImage(_ url: URL, completion: @escaping (UIImage?) -> Void) {
        let task = URLSession.shared.dataTask(with: url) { data, response, error in
            guard let data = data, error == nil else {
                DispatchQueue.main.async {
                    completion(nil)
                }
                return
            }
            let image = UIImage(data: data)
            DispatchQueue.main.async {
                completion(image)
            }
        }
        task.resume()
    }
}

// Utilisation :
// voir exmple précédent.
```

Démonstration `ImageLoader`

Nous allons faire une app qui gère des formats de réponses en JSON.

Vidéo : lien

PokemonDownloader

Vidéo: lien

Décodage JSON avec `JSONDecoder().decode(_:from:)`

Le décodage JSON en Swift est facilité par l'utilisation de la fonction `JSONDecoder().decode(_:from:)`. Cette fonction permet de convertir une instance de `Data` en une structure ou classe Swift décodable.

Voici une explication plus détaillée de son fonctionnement :

1. **Création d'une instance `JSONDecoder` :** `JSONDecoder` est une classe qui décode un objet JSON en une instance de valeur de type spécifique en Swift. Vous créez une instance en appelant `JSONDecoder()`.
2. **Appel de la fonction `decode(:from:)` : La fonction `decode(:from:)`** est une fonction générique, ce qui signifie qu'elle fonctionne avec n'importe quel type de Swift qui est conforme à la protocole `Decodable`. Vous spécifiez le type de la valeur que vous attendez à décoder en passant

le type comme premier argument. Par exemple, si vous attendez un tableau de strings, vous utilisez `Array<String>.self` comme premier argument.

3. **Passer les données JSON à décoder :** Le deuxième argument de la fonction `decode(_:from:)` est l'instance de `Data` contenant les données JSON à décoder. En général, ces données proviennent d'une réponse de serveur web.
4. **Gestion des erreurs :** Comme le processus de décodage peut échouer (par exemple, si les données JSON ne correspondent pas au modèle de données Swift ou si les données sont mal formées), l'appel à `decode(_:from:)` doit être placé dans un bloc `do-catch` pour gérer les erreurs.

```
struct User: Decodable {
    let id: Int
    let name: String
}

let jsonData = """
{
    "id": 123,
    "name": "John Doe"
}
""".data(using: .utf8)!

do {
    let user = try JSONDecoder().decode(User.self, from: jsonData)
    print("ID: \(user.id), Name: \(user.name)")
} catch {
    print("Error decoding JSON: \(error)")
}
```

Dans cet exemple, `User.self` est le type de valeur que nous attendons à décoder, et `jsonData` est l'instance de `Data` contenant les données JSON à décoder. Si le décodage est réussi, nous avons alors accès à un objet `User` que nous pouvons utiliser dans notre code.

Extrait de code

```
import Foundation

struct Pokemon: Decodable {
    let name: String
    let id: Int
    let sprites: Sprites
    var imageURL : String {
        return self.sprites.front_default
    }
}

struct Sprites: Decodable {
    let front_default : String
```

```
}



class PokemonDownloader {
    // Singleton instance
    static let shared = PokemonDownloader()

    private init() {}

    func downloadPokemonData(for name: String, completion: @escaping (Pokemon?) ->
Void) {
        let urlRef = "https://pokeapi.co/api/v2/pokemon/\(name)"
        guard let url = URL(string: urlRef) else {
            print("Invalid URL: \(urlRef)")
            return
        }

        let task = URLSession.shared.dataTask(with: url) { (data, response, error) in
            if let error = error {
                print("Error: \(error)")
            } else if let data = data {
                let decoder = JSONDecoder()
                do {
                    let pokemon = try decoder.decode(Pokemon.self, from: data)
                    DispatchQueue.main.async {
                        completion(pokemon)
                    }
                } catch {
                    print("Error decoding data: \(error)")
                }
            }
        }

        task.resume()
    }
}
```

Extrait du fichier `ViewController.swift`

```
//...
let pokemons = ["pikachu", "ditto", "charizard", "poliwrath"]

//...

class ViewController: UIViewController {
//...
    @objc func click() {
        PokemonDownloader.shared.downloadPokemonData(for: pokemons.randomElement()!) {
pokemon in
```

```
            guard let pokemon else {
                print("Error: Invalide data")
                return
            }
            ImageDownloader.shared.downloadImage(URL(string: pokemon.imageURL)!) {
image in
                self.imageView.image = image
            }
            self.captionLabel.text = "ID: \(pokemon.id) - Nom: \(pokemon.name)"
        }
    }

    //...
}
```