Networking (APIClient.swift)

Introducción

El código del networking se encuentra en el archivo `APIClient.swift` y se encarga de manejar las solicitudes de red a la API de Rick y Morty. Esta documentación desglosa los componentes clave del código y explica su funcionamiento en detalle.

1. Clase `APIClient`

La clase `APIClient` sirve como punto de acceso para realizar peticiones a la API. Es responsable de construir las URLs de las solicitudes, gestionar las respuestas y entregar los datos obtenidos a la capa de vista de la aplicación.

1.1 Propiedades:

- `baseURL`: Almacena la URL base de la API de Rick y Morty. Esta URL se utiliza para construir las URLs completas para las solicitudes.

1.2 Método `init`

- El método de inicialización `init` crea una instancia de `APIClient` tomando la URL base como parámetro. Guarda esta URL en la propiedad `baseURL` para su uso posterior.

1.3 Método `fetchCharacters`

- El método `fetchCharacters` es la función principal para obtener datos de personajes de la API. Realiza las siguientes tareas:

    1. **Construye la URL de la solicitud:** Combina la URL base (`baseURL`) con el endpoint `/character` para obtener la URL completa de la solicitud.

    2. **Crea una `URLRequest`:** Inicializa una `URLRequest` con la URL construida en el paso anterior. Configura el método HTTP a `GET`, indicando que se trata de una solicitud para recuperar datos.

    3. **Realiza la solicitud de red:** Utiliza `URLSession.shared.dataTask(with:completion:)` para crear una tarea de datos asociada a la `URLRequest`. La tarea se ejecuta en segundo plano y llama al manejador de finalización cuando se recibe la respuesta o se produce un error.

    4. **Maneja la respuesta:** Dentro del manejador de finalización, se comprueba si se ha producido un error. Si hay un error, se llama al manejador de finalización con un `Result.failure` que contiene el error.

    5. **Valida la respuesta HTTP:** Si no hay error, se verifica que la respuesta HTTP tenga un código de estado 200 (OK). Si el código de estado no es 200, se crea un error personalizado y se llama al manejador de finalización con un `Result.failure` que contiene este error.

6. **Decodifica los datos JSON:** Si la respuesta HTTP es válida, se extraen los datos JSON del cuerpo de la respuesta. Se utiliza `JSONDecoder` para decodificar los datos JSON en un objeto `CharacterResponse`.

7. **Llama al manejador de finalización con éxito:** Si la decodificación de JSON es exitosa, se llama al manejador de finalización con un `Result.success` que contiene el objeto `CharacterResponse` decodificado.

2. Manejo de Errores

El código de networking implementa un manejo de errores básico para las siguientes situaciones:

- **Errores de red:** Si la solicitud falla debido a problemas de red, como la falta de conectividad o un error de tiempo de espera, se captura el error y se llama al manejador de finalización con un `Result.failure` que contiene el error de red.

- **Códigos de estado HTTP no válidos:** Si la respuesta HTTP tiene un código de estado que no es 200 (OK), se crea un error personalizado con un mensaje descriptivo y se llama al manejador de finalización con un `Result.failure`que contiene este error personalizado.

- **Errores de decodificación JSON:** Si la decodificación de los datos JSON en un objeto `CharacterResponse` falla, se captura el error y se llama al manejador de finalización con un `Result.failure` que contiene el error de decodificación.

3. Mejoras Posibles

- **Almacenamiento en caché de respuestas:** Para mejorar el rendimiento, se puede implementar el almacenamiento en caché de las respuestas exitosas de la API para evitar solicitudes innecesarias. Esto se puede lograr utilizando mecanismos como `URLCache` o bibliotecas de terceros como Kingfisher o SDWebImage.

- **Manejo de errores más específico:** Se puede ampliar el manejo de errores para identificar y manejar diferentes tipos de errores con mayor precisión, proporcionando información más útil al desarrollador o al usuario.

- **Uso de `URLSession.shared.dataTask(with:completion:)`:** La función `URLSession.shared.dataTask(with:completion:)` se utiliza para realizar solicitudes de red de forma asíncrona. Esta función permite que la aplicación continúe ejecutando

SwiftUI ContentView (ContentView.swift)

Introducción

El archivo `ContentView.swift` contiene la estructura `ContentView`, que es el punto de entrada principal de la aplicación SwiftUI. Esta vista se encarga de mostrar la lista de personajes obtenidos de la API de Rick y Morty, así como de manejar las interacciones del usuario, como la navegación a la vista de detalles de un personaje seleccionado.

Componentes Clave

1. **Gestión de Estado:**
    - `@State private var characters: [Character] = []`: Esta propiedad almacena la matriz de objetos `Character` que representan los datos de los personajes obtenidos de la API.
    - `@State private var error: Error?`: Esta propiedad almacena cualquier error que pueda surgir durante la solicitud de datos o el manejo de la información.
2. **Cuerpo de la Vista:**
    - `NavigationView`: Encapsula la vista principal y proporciona una barra de navegación.
    - `List(characters, id: \.id)`: Muestra una lista de personajes utilizando la vista `List`. El identificador único para cada personaje se establece en `\.id`.
        - `NavigationLink(destination: CharacterDetailView(character: character))`: Permite la navegación a la vista `CharacterDetailView` cuando se selecciona un personaje.
            - `Image(uiImage: UIImage(data: try! Data(contentsOf: URL(string: character.image)!)) ?? UIImage())`: Carga la imagen del personaje utilizando la URL proporcionada en la propiedad `image` del objeto `Character`.
            - `Text(character.name)`: Muestra el nombre del personaje.
        - `listRowBackground`: Aplica un fondo de color a las filas de la lista.
    - `.navigationTitle("Mostrar Personajes")`: Establece el título de la barra de navegación.
    - `.font(.custom("AmericanTypewriter", fixedSize: 24))`: Personaliza la fuente y el tamaño del texto para el título.

- `.foregroundColor(Color.yellow)`: Establece el color de texto del título en amarillo.
- `.onAppear { fetchCharacters() }`: Ejecuta la función `fetchCharacters` cuando la vista aparece por primera vez para cargar los datos de los personajes.
- `.alert(isPresented: Binding<Bool>(get: { self.error != nil }, set: { _ in self.error = nil })) { Alert(title: Text("Error"), message: Text(error!.localizedDescription), dismissButton: .default(Text("OK"))) }`: Muestra una alerta en caso de error, mostrando el mensaje de error correspondiente.
- `.scrollContentBackground(.hidden)`: Oculta el fondo predeterminado de la vista de desplazamiento.

3. **Función `fetchCharacters`:**
   - Esta función se encarga de obtener los datos de los personajes de la API de Rick y Morty.
   - Utiliza la clase `APIClient` para realizar la solicitud de red.
   - Actualiza la propiedad `characters` con los datos obtenidos de la API.
   - Maneja los errores que puedan surgir durante la solicitud o el manejo de la información.

Mejoras Posibles

- **Almacenamiento en caché de imágenes:** Se puede implementar el almacenamiento en caché de las imágenes de los personajes para mejorar el rendimiento y reducir la carga de la red. Esto se puede lograr utilizando mecanismos como `URLCache` o bibliotecas de terceros como Kingfisher o SDWebImage.
- **Manejo de errores más específico:** Se puede ampliar el manejo de errores para identificar y manejar diferentes tipos de errores con mayor precisión, proporcionando información más útil al desarrollador o al usuario.
- **Uso de `CachedAsyncImage`:** Si está disponible en la versión de SwiftUI que se utiliza, se puede reemplazar la carga manual de imágenes con `CachedAsyncImage`. Esta función proporciona un manejo automático del almacenamiento en caché y la carga de imágenes.
- **Implementación de búsqueda:** Se puede agregar una función de búsqueda para permitir a los usuarios encontrar personajes específicos por nombre o por otros criterios.
- **Paginación de datos:** Si la API devuelve una gran cantidad de datos, se puede implementar la paginación para mostrar los personajes en páginas, cargando más datos a medida que el usuario se desplaza por la lista.

Mejora del Código con Caché

Introducción

Para mejorar el rendimiento de la aplicación y reducir la carga de la red, se puede implementar el almacenamiento en caché de las imágenes de los personajes y las respuestas de la API. Esto significa que la aplicación guardará temporalmente las imágenes y los datos obtenidos de la API, para evitar tener que volver a descargarlos cada vez que se accede a ellos.

1. Caché de Imágenes

- **Utilizar `URLCache`:**
    - **`URLCache.shared.setCachedResponse(data: response, forRequest: request)`:** Se puede utilizar esta función para almacenar la respuesta de una solicitud de red en el caché. La respuesta se asocia con la solicitud específica (`request`) que se utilizó para obtenerla.
    - **`URLCache.shared.cachedResponse(forRequest: request)`:** Esta función recupera la respuesta almacenada en caché para una solicitud específica. Si la respuesta no está en caché, se devuelve `nil`.
- **Bibliotecas de Terceros:**
    - **Kingfisher:** Esta biblioteca proporciona una forma eficiente de cargar y almacenar en caché imágenes. Ofrece funciones para cargar imágenes desde URL, almacenarlas en caché en el disco o en la memoria, y manejar errores de red.
    - **SDWebImage:** Otra biblioteca popular para cargar y almacenar en caché imágenes. Similar a Kingfisher, ofrece funciones para cargar, almacenar en caché y manejar errores de red.

2. Caché de Respuestas de la API

- **`UserDefaults`:**
    - **`UserDefaults.standard.set(data: data, forKey: key)`:** Se puede utilizar esta función para almacenar datos en `UserDefaults`, que es un sistema de almacenamiento en caché integrado en iOS. La clave (`key`) se utiliza para identificar los datos almacenados.
    - **`UserDefaults.standard.data(forKey: key)`:** Esta función recupera los datos almacenados en `UserDefaults` para una clave específica. Si no hay datos asociados con la clave, se devuelve `nil`.
- **Bibliotecas de Terceros:**
    - **Realm:** Una base de datos móvil que proporciona un modelo de objetos flexible y eficiente para almacenar datos estructurados.

- **Core Data:** Un marco de almacenamiento de datos integrado en iOS que permite almacenar y recuperar objetos de forma relacional.

Implementación en el Código

1. Caché de Imágenes

- **Cargar imágenes con Kingfisher o SDWebImage:** Reemplazar la carga manual de imágenes con `Kingfisher` o `SDWebImage`. Estas bibliotecas manejan automáticamente la descarga y el almacenamiento en caché de las imágenes.
- **Ejemplo con Kingfisher:**

Swift
```
import Kingfisher

Image(uiImage: {
  if let cachedImage = Kingfisher.shared.cache.image(forKey:
character.image) {
    return cachedImage
  }

  do {
    let data = try Data(contentsOf: URL(string:
character.image)!)
    return UIImage(data: data)
  } catch {
    return nil
  }
})
```

2. Caché de Respuestas de la API

- **Almacenar la respuesta de la API en `UserDefaults`:** Guardar la respuesta decodificada de la API (`CharacterResponse`) en `UserDefaults` utilizando una clave única.
- **Ejemplo con `UserDefaults`:**

Swift
```
private func cacheAPIResponse(responseObject:
CharacterResponse) {
  do {
    let data = try JSONEncoder().encode(responseObject)
    UserDefaults.standard.set(data, forKey:
"charactersCacheKey")
  } catch {
```

```
    print("Error al almacenar en caché la respuesta de la
API: \(error)")
  }
}
```

- **Recuperar la respuesta de la API de `UserDefaults`:** Antes de realizar una nueva solicitud a la API, verificar si la respuesta está almacenada en caché. Si lo está, decodificarla y utilizarla.

- **Ejemplo con `UserDefaults`:**

Swift

```
private func fetchCharacters() {
  if let cachedData = UserDefaults.standard.data(forKey:
"charactersCacheKey") {
    do {
      let responseObject = try
JSONDecoder().decode(CharacterResponse.self, from:
cachedData)
      self.characters = responseObject.results
      return
    } catch {
      print("Error al decodificar la respuesta de la API en
caché: \(error)")
    }
  }

  // Realizar la solicitud a la API si la respuesta no está
en caché
  // ...
}
```

1. Principio de Responsabilidad Única (SRP):

- **Parcialmente Cumplido:** La clase tiene la responsabilidad principal de obtener personajes de la API. Sin embargo, también crea objetos de error para errores de red y API. Si bien el manejo de errores está relacionado con las llamadas a la red, crear objetos de error separados podría considerarse una responsabilidad independiente.

2. Principio Abierto/Cerrado (OCP):

- **Parcialmente Cumplido:** Actualmente, el código obtiene personajes. Si necesita extender la funcionalidad para obtener otros datos (por ejemplo, ubicaciones, episodios), es posible que deba modificar la función `fetchCharacters` o crear una nueva función. Idealmente, la clase debería diseñarse para adaptarse a futuras extensiones para obtener diferentes tipos de datos sin modificar el código existente.

3. Principio de Sustitución de Liskov (LSP):

- **No aplicable:** Este principio se aplica a las jerarquías de herencia. Dado que `APIClient` no tiene subclases en este fragmento de código, LSP no es directamente relevante aquí.

4. Principio de Segregación de la Interfaz (ISP):

- **No aplicable:** Este principio se centra en dividir interfaces grandes en otras más pequeñas y específicas. El código no utiliza ninguna interfaz formal, por lo que ISP no es directamente aplicable aquí.

5. Principio de Inversión de Dependencia (DIP):

- **Parcialmente Cumplido:** El código depende de implementaciones concretas como `URLSession.shared` para las llamadas a la red. Idealmente, podría depender de una abstracción como un protocolo `NetworkClient` que defina el comportamiento de la comunicación de red. Esto permitiría realizar pruebas más fácilmente y el uso potencial de diferentes bibliotecas de red en el futuro.

Aquí hay algunas sugerencias de mejora basadas en los principios SOLID:

1. **SRP:** Considere crear una clase o función separada para el manejo de errores, especialmente si los objetos de error se vuelven más complejos.
2. **OCP:** Diseñe la clase teniendo en cuenta posibles extensiones futuras. Es posible que desee tener una función `fetchData` genérica con un parámetro que especifique el tipo de datos que se va a obtener.
3. **DIP:** Introduzca una abstracción como un protocolo `NetworkClient` para desacoplar el código de las implementaciones de red concretas.

Soluciones para cumplir con los principios SOLID en APIClient.swift

1. Principio de Responsabilidad Única (SRP):

Solución: Crear una clase o función separada para el manejo de errores, llamada `ErrorManager` o similar. Esto encapsularía la lógica de creación y manejo de errores, separándola de la responsabilidad de `APIClient` de obtener datos.

Ejemplo:

Swift
```
class ErrorManager {
  static func manejarErrorDeRed(error: Error) {
    // Implementa la lógica de manejo de errores de red
  }

  static func manejarErrorAPI(códigoEstado: Int) {
    // Implementa la lógica de manejo de errores de la API
  }
}
```

2. Principio Abierto/Cerrado (OCP):

Solución: Introducir una función genérica `fetchData` con un parámetro que especifique el tipo de datos a obtener. Esto permitiría obtener diferentes tipos de datos sin modificar la clase `APIClient`.

Ejemplo:

Swift
```
class APIClient {
  let baseURL: URL

  init(baseURL: URL) {
    self.baseURL = baseURL
  }

  func fetchData<T: Decodable>(ruta: String, completion:
@escaping (Result<T, Error>) -> Void) {
    let url = baseURL.appendingPathComponent(ruta)

    var request = URLRequest(url: url)
    request.httpMethod = "GET"

    URLSession.shared.dataTask(with: request) { data,
response, error in
      // ... (Lógica de manejo similar a fetchCharacters)

      do {
        let decoder = JSONDecoder()
        let objetoRespuesta = try decoder.decode(T.self,
from: data!)
        completion(.success(objetoRespuesta))
      } catch {
```

```
        completion(.failure(error))
      }
    }.resume()
  }
}
```

3. Principio de Inversión de Dependencia (DIP):

Solución: Implementa un protocolo `NetworkClient` para definir el comportamiento de comunicación de red y usa inyección de dependencia para proporcionar una instancia de `NetworkClient` a `APIClient`. Esto desacoplaría a `APIClient` de las implementaciones concretas de red.

Ejemplo:

Swift

```
protocol NetworkClient {
  func fetchData(url: URL, completion: @escaping
(Result<Data, Error>) -> Void)
}

class APIClient {
  let baseURL: URL
  let networkClient: NetworkClient

  init(baseURL: URL, networkClient: NetworkClient) {
    self.baseURL = baseURL
    self.networkClient = networkClient
  }

  func fetchData<T: Decodable>(ruta: String, completion:
@escaping (Result<T, Error>) -> Void) {
    let url = baseURL.appendingPathComponent(ruta)
    networkClient.fetchData(url: url) { result in
      // ... (Manejar la respuesta de red y decodificar en T)
    }
  }
}
```

4. Principio de Segregación de la Interfaz (ISP):

Solución: Como el código no utiliza interfaces formales, ISP no es directamente aplicable aquí. Sin embargo, si se introduce interfaces en el futuro, asegurar de que no sean demasiado grandes y representen funcionalidades específicas.

5. Principio de Sustitución de Liskov (LSP):

Solución: LSP no es directamente aplicable aquí ya que el código no tiene jerarquías de herencia. Sin embargo, si se introduces subclases en el futuro, asegurar de que se adhieran a LSP al no romper los contratos de sus clases base.