Preguntas de la entrevista de Swift para desarrolladores sénior de iOS (edición 2025)

1. **Diferencias entre class, struct y actor en Swift**

class (Tipo de referencia)

- Almacenado en el montón: los objetos viven en la memoria hasta que no quedan referencias.
- Paso por referencia: cuando se asigna a una nueva variable, solo se asigna la referencia (puntero) copiado, no los datos.
- Admite herencia: las clases pueden heredar propiedades y métodos de otra clase.
- Mutable incluso si se declara con let: A diferencia de las estructuras, las propiedades de una clase pueden ser modificado incluso si la instancia está asignada a una constante.

Ejemplo:

```
class Person {
        var name: string
        init(name: string) {
                Self.name = name
        }
}

let person1 = Person(name: "Alicia")
let person2 = person1 // Tanto person1 como person2 hacen referencia a la misma dirección de memoria
person2.name = "Bob"

print(person1.name) // "Bob" (Los cambios se reflejan en ambos casos)
```

struct (Tipo de valor – Value Type)

- Almacenado en la pila: Las estructuras se destruyen cuando salen del alcance, lo que las hace más eficientes en la memoria.
- Pasar por valor: Al asignar una estructura a una nueva variable, se copian los datos.
- Sin herencia: Las estructuras no admiten la herencia.
- Control de mutabilidad: Requiere la palabra clave mutante para modificar las propiedades de los métodos.

Ejemplo:

```
struct Person {
        var name: String
}

var person1 = Person(name: "Alicia")
var person2 = person1 // Crea una copia separada

person2.name = "Bob"
print(person1.name) // "Alice" (Sin cambios en la instancia original)
```

actor (Tipo de referencia con simultaneidad – Reference Type with Concurrency)

- Garantiza la seguridad de los hilos: Protege los datos de las condiciones de carrera.
- Acceso secuencial: permite que solo una tarea acceda a las propiedades a la vez.

Ejemplo:

```
actor BankAccount {
```

```
    private var balance: Int = 0

    func deposit(amount: Int) {
        balance += amount
    }

    func getBalance() -> Int {
        return balance
    }
}

let account = BankAccount()
Task {
    await account.deposit(amount: 100)
    print(await account.getBalance()) // Garantiza un acceso seguro
}
```

2. **Envoltorios de propiedades en Swift (Property Wrappers)**

Los contenedores de propiedades permiten lógica reutilizable para las propiedades.

Creación de un contenedor de propiedades personalizado

```
@propertyWrapper
struct Capitalized {
    private var value: String = ""

    var wrappedValue: String {
        get { value }
        set { value = newValue.capitalized}
    }
}

struct User {
    @Capitalized var name: String
}

var user = User(name: "john doe")
print(user.name) // "John Doe"
```

- **Caso de uso:** Capitalizar automáticamente la entrada del usuario.

3. **@autoclosure y su uso**

@autoclosure permite que las expresiones se conviertan automáticamente en cierres.

Sin @autoclosure

```
func logIfTrue(_ condition: () -> Bool) {
    if condition() {
        print("Condición cumplida")
    }
}

logIfTrue({ 5 > 3 }) // Cierre explícito
```

Con @autoclosure

```
func logIfTrue(_ condition: @autoclosure () -> Bool) {
	if condition() {
		print("Condición cumplida")
	}
}

logIfTrue(5 > 3) // No es necesario envolver '{ }'
```

- **Caso de uso:** Evite cálculos innecesarios (por ejemplo, aserciones, registro).

4. **Gestión de memoria de Swift y ARC**
   - **El recuento automático de referencias (ARC)** rastrea referencias a instancias de clase.
   - Los objetos son desasignados automáticamente cuando el recuento de referencias = 0

Ejemplo:

```
clase Car {
	var model: String
	init(model: String) {
		self.model = model
		print("\(model) inicializado")
	}

	deinit {
		print("\(model) desasignado")
	}
}

var car1: Car? = Car(model: "Tesla")
car1 = nil // "Tesla desasignado"
```

- **Caso de uso:** Manejo eficiente de memoria en Swift.

5. **Retención de ciclos y prevención**

Un ciclo de retención ocurre cuando dos objetos hacen una fuerte referencia entre sí.

Ejemplo de ciclo de retención:

```
class Person {
	var name: String
	var pet: Pet? // Referencia fuerte
	init(name:String) { self.name = name }
}

class Pet {
	var name: String
	var owner: Person? // Referencia fuerte
	init(name:String) { self.name = name }
}
```

Solución: Usar weak o unowned

```
class Person {
	var name: String
	weak var pet: Pet? // La referencia débil evita el ciclo de retención
}
```

6. **Parámetros inout en Swift**

Permite modificar los parámetros de la función directamente.

```
func doubleValue(_ number: inout Int) {
	number *= 2
}

var value = 10
doubleValue(&value)
print(value) // 20
```

- **Caso de uso:** Modificación de variables sin devolver un nuevo valor.

7. **Codable y Decodable con manejo JSON personalizado**

Swift puede codificar y decodificar JSON usando Codable.

Ejemplo básico:

```
struct User: Codable {
	var name: String
	var age: Int
}
```

Ejemplo de decodificación personalizada:

```
struct User: Codable {
	var name: String
	var age: Int

	enumCodingKeys: String, CodingKey {
		case name
		case age = "user_age" // Asigna la clave JSON a una propiedad diferente nombre
	}
}
```

- **Caso de uso:** Manejo de diferentes formatos JSON.

8. **Cierres con escape vs sin escape (Escaping vs Non-Escaping Closures)**

Sin escape (predeterminado)

```
func performTask(task: () -> Void) {
	task() // Se ejecuta inmediatamente
}
```

Escape (@escaping)

```
func performAsyncTask(completion: @escaping () -> Void) {
	DispatchQueue.global().async {
		sleep(2)
		completion()
	}
}
```

- **Caso de uso:** Operaciones asíncronas.

9. Constructores de funciones en Swift

Se utiliza en SwiftUI para construir estructuras complejas.

```
@resultBuilder
struct StringBuilder {
	static func buildBlock(_ components: String...) -> String {
		components.joined(separator: " ")
	}
}

func makeSentence(@StringBuilder _ content: () -> String) -> String {
	content()
}

let sentence = makeSentence {
	"Swift"
	"es"
	"¡increíble!"
}

print(sentence) // "¡Swift es increíble!"
```

- **Caso de uso:** DSLs como SwiftUI.

10. Any vs AnyObject vs AnyHashable

- Any: Cualquier tipo ((class, struct, enum, etc).
- AnyObject: Solo tipos de clase.
- AnyHashable: Cualquier tipo que se ajuste a Hashable.

```
let anything: Any = 42
let anyObject: AnyObject = NSString("Hello")
let hashable: AnyHashable = "Swift"
```

- **Caso de uso:** API flexibles que aceptan varios tipos.

11. Diferencia entre herencia de protocolo y clase

Herencia de clases:

- Herencia única: Una clase puede heredar de una sola superclase.
- Administración de estado: Puede tener propiedades almacenadas.
- Tipo de referencia: La modificación de una instancia afecta a todas las referencias.

- Envío dinámico: Los métodos se pueden anular mediante la anulación y dinámicamente enviado en tiempo de ejecución.

Ejemplo:

```
class Animal {
	var name: String
	init(name: String) { self.name = name }

	func makeSound() {
		print("Algún sonido animal genérico")
	}
}

class Dog: Animal {
	override func makeSound() {
		print("Guau!")
	}
}

let dog = Dog(name: "Buddy")
dog.makeSound() // "¡Guau!"
```

Herencia de protocolo:

- Herencia múltiple: Un tipo puede ajustarse a varios protocolos.
- Sin propiedades almacenadas: Solo define el comportamiento.
- Envío estático: Los métodos de los protocolos generalmente se resuelven en tiempo de compilación.

Ejemplo:

```
protocol Flyable {
	func fly()
}

protocol Swimmable {
	func swim()
}

class Duck: Flyable, Swimmable {
	func fly() {
		print("El pato está volando")
	}

	func swim() {
		print("El pato está nadando")
	}
}
```

🔹 **Conclusión clave:**

- Use la herencia de clases para el estado compartido.
- Utilice protocolos para la composición del comportamiento.

12. Tipos asociados en protocolos Swift

- associatedtype permite que un protocolo defina un tipo de marcador de posición.
- El tipo real está determinado por el tipo conforme.

Ejemplo:

```
protocol Container {
	associatedtype Item
	func add(_ item: Item)
	func getAll() -> [Item]
}

class IntContainer: Container {
	private var items: [Int] = []

	func add(_ item: Int) { items.append(item) }
	func getAll() -> [Int] { items }
}
```

- **¿Por qué?**
  - Aumenta la flexibilidad.
  - Permite definir un comportamiento similar al genérico dentro de los protocolos.

13. Programación orientada a protocolos (POP) vs Programación orientada a objetos (POO)

Enfoque OOP:

- Usa la herencia para compartir el comportamiento.
- A menudo resulta en jerarquías de clases profundas

Ejemplo:

```
class Animal {
	func makeSound() { print("Algún sonido de animal") }
}

class Dog: Animal {
	override func makeSound() { print("Bark!") }
}
```

Enfoque POP:

- Usa extensiones de protocolo para compartir comportamiento.
- Fomenta la composición sobre la herencia.

Ejemplo:

```
protocol SoundMaking {
	func makeSound()
}

extensión SoundMaking {
	func makeSound() { print("Sonido predeterminado") }
}

struct Dog: SoundMaking {
```

```
    func makeSound() { print("Ladrido!") }
}
```

- **POP evita jerarquías profundas, lo que hace que el código sea más modular y flexible.**

14. Propósito del Self en los protocolos

Self se refiere al tipo conforme dentro de un protocolo.

Ejemplo: Retornando Self

```
protocol Cloneable {
    func clone() -> Self
}

class Car: Clonable {
    func clone() -> Self { return self }
}
```

- **¿Por qué? Asegura que clone() siempre devuelva el tipo correcto.**

15. Restricciones genéricas en Swift (Generic Constraints)

Restringir un tipo genérico a un requisito específico.

Ejemplo: Genérico con una restricción

```
func findLargest<T: Comparable>(in array: [T]) -> T? {
    return array.max()
}

print(findLargest(en: [1, 2, 3, 4])) // 4
```

Usando clausula where

```
func findElement<T>(in array: [T]) where T: Equatable {
    // Cuerpo de la función
}
```

- **¿Por qué? Garantiza la seguridad del tipo al tiempo que permite flexibilidad.**

16. Ampliación de un protocolo

Las extensiones de protocolo permiten agregar implementaciones predeterminadas.

Ejemplo:

```
protocol Greetable {
    func greet()
}

extension Greetable {
    func greet() {
        print("¡Hola, mundo!")
    }
}
```

```
struct Person: Greetable {}
let person = Person()
person.greet() // "¡Hola, mundo!"
```

- **Reduce el código repetitivo al proporcionar implementaciones predeterminadas.**

17. Tipos de retorno opacos (some) en Swift

Oculta los tipos de retorno de concreto al tiempo que garantiza la seguridad del tipo.

Ejemplo:

```
protocol Shape {
        func area() -> Double
}

struct Circle: Shape {
        var radius: Double
        func area() -> Double { return .pi * radio * radio }
}

func getShape() -> some Shape { return Circle(radius: 5) }
```

- **¿Por qué?**
  - Evita exponer los detalles de implementación.
  - Hace que los tipos de valor devuelto sean más flexibles.

18. Covarianza y contravarianza en genéricos Swift

Covarianza (Subtipo permitido)

- Permite un subtipo en el que se espera un supertipo.

```
class Animal {}
class Dog: Animal {}
let animals: [Animal] = [Dog()] // ☑ Permitido (El Array es covariante)
```

Contravarianza (Supertipo permitido)

- Permite un supertipo donde se espera un subtipo.

```
func printAnimalInfo(_ animal: Animal) {
        print("Animal info")
}

let dogPrinter: (Dog) -> Void = printAnimalInfo // ☑ Permitido (Contravariante)
```

- **¿Por qué? Ayuda a mantener la seguridad de tipos.**

19. Manejo de múltiples conformidades de protocolo con métodos en conflicto

Cuando dos protocolos definen el mismo método, Swift requiere una desambiguación explícita.

Ejemplo:

```
protocol A {
	func greet()
}

protocol B {
	func greet()
}

struct MyStruct: A, B {
	func greet() { print("Hola desde MyStruct") }
}
```

Solución: Usar extensiones

```
extension MyStruct: A {
	func greet() { print("Hola de A") }
}

extension MyStruct: B {
	func greet() { print("Hola desde B") }
}
```

- **¿Por qué? Garantiza la resolución correcta del método.**

20. @objc y por qué es necesario

- Se utiliza para exponer métodos Swift al tiempo de ejecución de Objective-C.
- Necesario para las APIs basadas en selector (por ejemplo, #selector()).

Ejemplo:

```
import Foundation

class MyClass: NSObject {
	@objc func sayHello() {
		print("¡Hola, Objective-C!")
	}
}

let selector = #selector(MyClass.sayHello)
```

- **¿Por qué?**
  - Permite la compatibilidad con frameworks Objective-C.
  - Permite llamadas a métodos usando #selector().

21. Explicar la diferencia entre Grand Central Dispatch (GCD) y las colas de operaciones (Operation Queues).

Grand Central Dispatch (GCD) y **Operation Queues (NSOperationQueue)** son mecanismos de simultaneidad en iOS, pero tienen diferencias clave en cuanto a flexibilidad y complejidad:

Características	GCD (DispatchQueue)	Operation Queues (NSOperationQueue)
Nivel de abstracción	API de bajo nivel (basada en C)	API de alto nivel (basada en Objective-C)

Representación de tareas	Utiliza Closures (DispatchWorkItem)	Utiliza subclases NSOperation
Control de dependencias	Sin administración de dependencias incorporada	Admite dependencias entre operaciones
Cancelación de tareas	Sin cancelación directa (debe manejarse manualmente)	Admite el método cancel() para NSOperation
Control de prioridad	Utiliza QoS (Calidad de servicio)	Control de prioridad más avanzado con dependencias
Observancia	Sin soporte directo de KVO	Admite KVO (Observación de clave-valor)
Thread Safety	Herramientas básicas de sincronización	Más robusto con dependencias y Estados de ejecución

- Usa **GCD** cuando necesites una simultaneidad simple, liviana y de baja sobrecarga.
- Usa **Operation Queues** cuando necesite administración, cancelación y observabilidad de dependencias.

22. ¿Cuáles son las diferencias entre async/await, DispatchQueu y NSOperationQueue?

Característica	async await	DispatchQueue (GCD)	NSOperationQueue
Nivel de abstracción	Alto nivel (Swift concurrencia)	Nivel medio (GCDAPI)	Alto nivel (API de base)
Concurrencia y Modelo	Concurrencia estructurada	Basado en tareas, depacho de colas	Orientada a objetos, colas de operaciones
Sintaxis	async let, await	DispatchQueue.async {}	OperationQueue.addOperation {}
Cancelación	Soporta cancelación cooperativa (Task.cancel())	Sin cancelación incorporado	Admite cancel() en NSOperation
Manejo de dependencia	No soportado directamente (gestionado por TaskGroup). Optimizado para Swift.	Sin dependencias	Admite dependencias a través de addDependency()
Rendimiento	Concurrencia estructurada	Ligero, eficiente para tareas sencillas	Más gastos generales debido a Diseño orientado a objetos

- Uso de async/await para la concurrencia estructurada y segura con cancelación cooperativa.
- Uso de GCD (DispatchQueue) para una simultaneidad simple sin dependencias.
- Uso de NSOperationQueue para dependencias complejas y administración de tareas.

23. ¿Qué es un actor en Swift? ¿Cómo ayuda en la seguridad de los hilos?

Los actores en Swift son un tipo seguro de simultaneidad que serializa el acceso a su estado mutable, evitando condiciones de carrera.

```
actor BankAccount {
        private var balance: Int = 0

        func deposit(amount: Int) {
                balance += amount
```

```
    }

    func getBalance() -> Int {
        return balance
    }
}
```

Cómo los actores garantizan la seguridad de los hilos

1. **Encapsulación de estado:** Todas las propiedades dentro de un actor están protegidas de las carreras de datos.

Acceso asíncrono: Se debe acceder a los métodos dentro de un actor mediante await:

```
let account = BankAccount()
await account.deposit(amount: 100)
```

2. **Serialización automática:** Se serializarán varias tareas que llamen a un actor, lo que evitará escrituras simultáneas.

- Usa actores cuando necesites acceso simultáneo seguro al estado mutable compartido.

24. Explica Task, TaskGroup y DetachedTask en la simultaneidad de Swift.

1. Task (Simultaneidad estructurada)

- Ejecuta una operación asíncrona dentro del modelo de simultaneidad estructurado.
- Puede heredar prioridades de tareas.

Ejemplo:

```
Task {
    await fetchData()
}
```

2. TaskGroup (Ejecución paralela)

- Habilita tareas secundarias simultáneas dentro de un único ámbito primario.

Ejemplo:

```
await withTaskGroup(of: Int.self) { group in
    for i in 1...5 {
        group.addTask { return i * i }
    }
}
```

3. DetachedTask (simultaneidad no estructurada)

- Se ejecuta independientemente sin heredar la prioridad o el contexto de la tarea.

Ejemplo:

```
let handle = Task.detached {
    return await fetchData()
}

let result = await handle.value
```

- Usa Task para simultaneidad estructurada, TaskGroup para ejecución paralela y DetachedTask cuando se requiere la desasociación del contexto primario.

25. ¿Cómo funciona MainActor? ¿Cuándo deberías usarlo?

@MainActor garantiza que el código se ejecute en el subproceso principal.

Ejemplo:

```
@MainActor
class ViewModel {
	var title: String = "Hola"

	func updateTitle() {
		title = "Actualizado"
	}
}
```

O para una función:

```
@MainActor
func updateUI() {
	label.text = "Actualizado"
}
```

- Usa MainActor para actualizaciones de la interfaz de usuario y tareas enlazadas a subprocesos principales.

26. ¿Qué es la concurrencia estructurada y no estructurada en Swift?

Tipo	Definición	Ejemplo

Concurrencia estructurada	Las tareas se administran dentro de un alcance, loque garantiza que se completen antes de que se cierre la función.	`Task {`
		`  await fetchData()`
		`}`

Concurrencia no estructurada	Las tareas se ejecutan de forma independiente sin supervisión directa.	`Task.detached {`
		`  await fetchData()`
		`}`

- Uso de simultaneidad estructurada (Task, TaskGroup) para la seguridad y no estructurada simultaneidad (DetachedTask) cuando sea necesario.

27. Explicar las diferencias entre Task.sleep() y Thread.sleep().

Característica	Task.sleep()	Thread.sleep()
Modelo de concurrencia	Funciona con simultaneidad de Swift	Bloquea todo el hilo
Rendimiento	Sin bloqueo, suspende solo la tarea	Bloqueo, ineficaz para la simultaneidad
Uso	await Task.sleep(1_000_000_000)	Thread.sleep(forTimeInterval: 1.0)

- Usa Task.sleep() en la simultaneidad de Swift para evitar el bloqueo de subprocesos.

28. ¿Cómo maneja Swift las condiciones de carrera?

Swift evita las condiciones de carrera mediante:

1. **Actores** (@MainActor, actor).
2. **Mecanismos de aislamiento** (Task, TaskGroup).
3. **Colas de envío en serie** (DispatchQueue.serial).
4. **Cerraduras y semáforos** (si es necesario en el código heredado).

29. ¿Qué es el punto muerto (deadlock) y cómo puede prevenirlo?

Un interbloqueo se produce cuando dos subprocesos esperan indefinidamente a que el otro libere recursos.

Ejemplo de interbloqueo:

```
let queue = DispatchQueue(label: "com.deadlock")
queue.sync {
    queue.sync { print("Deadlock") } // Esto causará un interbloqueo
}
```

Cómo prevenir los interbloqueos:

1. Evite las llamadas de sincronización anidadas.
2. Utilice async en lugar de sync al enviar tareas.
3. Use mecanismos de tiempo de espera en bloqueos.

30. ¿Cómo manejaría una solicitud de red que requiere varias llamadas API dependientes de forma asíncrona?

Usa async/await con simultaneidad estructurada:

```
func fetchUserProfile() async throws -> UserProfile {
    let user = try await fetchUser()
    let posts = try await fetchPosts(user.id)
    let friends = try await fetchFriends(user.id)

    return UserProfile(user: user, posts: posts, friends: friends)
}
```

Usa async/await para dependencias y TaskGroup para la ejecución en paralelo.

31. Diferencias entre MVC, MVVM y VIPER

Estos son patrones arquitectónicos que se usan para organizar las aplicaciones iOS:

Model-View-Controller (MVC):

- **Estructura:**
  - Model: Administra datos y lógica de negocios.
  - View: Controla la representación de la interfaz de usuario.
  - Controller: Actúa como mediador entre el modelo y la vista.
- **Pros:** Simple y ampliamente utilizado.
- **Contras:** Conduce a "Controlador de vista masiva" debido a la lógica inflada.
- **Caso de uso:** Mejor para aplicaciones pequeñas o cuando se necesita un desarrollo rápido.

Model-View-ViewModel (MVVM):

- **Estructura:**
  - **Model:** Datos y lógica.
  - **View:** Representación de la interfaz de usuario.
  - **ViewModel:** Transforma los datos de View y controla la lógica de presentación.
- **Pros:** Mejor separación de preocupaciones, mejor capacidad de prueba.
- **Contras:** Requiere más configuración que MVC.
- **Caso de uso:** Mejor para aplicaciones medianas y grandes, especialmente cuando se usa SwiftUI (debido a bindings).

View-Interactor-Presenter-Entity-Router (VIPER):

- **Estructura:**
  - **View:** Capa de interfaz de usuario.
  - **Interactor:** Maneja la lógica de negocio.
  - **Presenter:** Prepara datos para View.
  - **Entity:** Capa de modelo.
  - **Router:** Maneja la navegación.
- **Pros:** Alta modularidad, excelente capacidad de prueba.
- **Contras:** Complejo y requiere más código.
- **Caso de uso:** Aplicaciones a gran escala con múltiples capas de lógica.

Elegir entre ellos

- **MVC** → Proyectos pequeños, creación rápida de prototipos.
- **MVVM** → Aplicaciones medianas/grandes, mejor capacidad de mantenimiento.
- **VIPER** → Aplicaciones empresariales en las que la modularidad y la escalabilidad son clave.

32. Uso de Combine para programación reactiva

Combine es el marco reactivo de Apple que permite la programación declarativa y funcional.

Ejemplo: Observación de la respuesta de la API

```
Import Combine

struct APIClient {
    func fetchData() -> AnyPublisher<String, error> {
        let url = URL(string: "https://api.example.com/data")!
        return URLSession.shared.dataTaskPublisher(for: url)
            .map { String(decodificación: $0.data, as: UTF8.self) }
            .mapError { $0 as Error }
```

```
                .eraseToAnyPublisher()
        }
}
```

Uso de Combine con SwiftUI

```
class ViewModel: ObservableObject {
        @Published var data: String = ""
        var cancellables = Set<AnyCancellable>()

        func loadData() {
                APIClient().fetchData()
                        .receive(on: DispatchQueue.main)
                        .sink(receiveCompletion: { _ in }, receiveValue: { self.data=$0 })
                        .store(in: &cancelables)
        }
}
```

Casos de uso:

- Manejo de respuestas de API.
- Actualizaciones de la interfaz de usuario en tiempo real (enlace de ViewModel a la interfaz de usuario).
- Encadenamiento de varias operaciones asincrónicas.

33. ObservableObject, @Published, y @State en SwiftUI

Estos se utilizan para manejar la administración de estado.

ObservableObject

Una clase que se ajusta a ObservableObject puede notificar a las vistas de SwiftUI sobre los cambios de estado.

```
class ViewModel: ObservableObject {
        @Published var count = 0
}
```

@Published

Se usa dentro de ObservableObject para notificar cuando cambia una propiedad.

```
class ViewModel: ObservableObject {
        @Published var username: String = ""
}
```

@State

Se utiliza para el estado local dentro de una vista de SwiftUI.

```
struct CounterView: View {
        @State private var count = 0
}
```

Diferencias:

Propiedad	Utilizado en	Alcance
@State	View	Local
@Published	ViewModel	Se utiliza con ObservableObject
ObservableObject	ViewModel	Estado global

34. Inyección de dependencias en Swift

La inserción de dependencias (DI) permite pasar dependencias en lugar de codificarlas de forma rígida.

1. Inyección de constructor

```
class NetworkService {
	func fetchData() { }
}

class ViewModel {
	let service: NetworkService

	init(service: NetworkService) {
		self.service = service
	}
}
```

2. Inyección de propiedades

```
clase ViewModel {
	var service: NetworkService?
}
```

3. Patrón Factory

```
class DependencyFactory {
	static func createService() -> NetworkService {
		return NetworkService()
	}
}
```

4. Uso de Swinject (DI Framework)

```
let container = Container()
container.register(NetworkService.self) { _ in NetworkService() }
```

Eligiendo el enfoque correcto:

- **Inyección de constructor:** Preferido para la inmutabilidad.
- **Inserción de propiedades:** se usa cuando la dependencia puede cambiar.
- **Patrón Factory:** Mejor para la gestión centralizada de dependencias.
- **Frameworks DI:** Adecuado para aplicaciones grandes.

35. Implementación de Deep Links en iOS

Los Deep Links permiten abrir pantallas de aplicaciones específicas a través de URL.

Uso de esquemas de URL:

1. Agregar un esquema de URL en Info.plist
2. Manejar en AppDelegate:

```
func application(_ app: UIApplication, open url: URL, options: [UIApplication.OpenURLOptionsKey: Any] = [:]) -> Bool {
        print("Abierto con URL: \(url)")
        return true
}
```

Uso de enlaces universales:

1. Configure los dominios asociados en Entitlements.plist.
2. Agregue un archivo apple-app-site-association (AASA) en el servidor.

36. Core Data vs Realm

Características	Core Data	Realm
Nativo de Apple	Si	No
Rendimiento	Bueno	Rápido
Migración de esquemas	Complejo	Fácil
Subprocesos múltiples	Difícil	Fácil
Manejo de relaciones	Complejo	Simple

¿Cuándo elegir?

- **Core Data:** Al integrarse con el ecosistema de Apple.
- **Realm:** Cuando el rendimiento y la facilidad de uso son prioridades.

37. Fuentes de datos diferenciables

DiffableDataSource simplifica el control de actualizaciones en UITableView/UICollectionView.

Tradicional UITableViewDataSource

```
func tableView(_ tableView: UITableView, cellForRowAt indexPath: IndexPath) -> UITableViewCell {
        // Creación de celdas estándar
}
```

Fuente de datos diferenciable

```
let dataSource = UITableViewDiffableDataSource<Section, Item>(tableView: tableView) {
tableView, indexPath, item in
        let cell = tableView.dequeueReusableCell(withIdentifier: "cell", for:indexPath)
        cell.textLabel?.text = item.title
        return cell
}
```

Ventajas:

- Animaciones automáticas
- No es necesario llamar manualmente reloadData().

38. Optimización del rendimiento del desplazamiento

- Usa cellForRowAt de manera eficiente.
- Implementa la reutilización de celdas (dequeueReusableCell).
- Usa vistas ligeras (drawRect con moderación).
- Habilita la captura previa (UITableViewDataSourcePrefetching).
- Utiliza subprocesos en segundo plano para el procesamiento de datos.

39. Integración de SwiftUI con UIKit

Integración de SwiftUI en UIKit

```
let swiftUIView = UIHostingController(rootView: MySwiftUIView())
```

Integración de UIKit en SwiftUI

```
struct MyUIKitView: UIViewControllerRepresentable {
        func makeUIViewController(context: Context) -> UIViewController {
                return MyViewController()
        }
}
```

40. ¿Qué son los App Clips y cómo se implementan?

Los App Clips son miniversiones ligeras, rápidas y enfocadas de una aplicación iOS que permiten a los usuarios acceder a funcionalidades específicas de la aplicación sin instalar la aplicación completa. Están diseñados para interacciones rápidas (menos de 10 MB) y se pueden iniciar a través de:

- Códigos QR
- Etiquetas NFC
- Banners inteligentes de Safari
- Mensajes y Mapsenlaces
- Códigos de App Clip (códigos visuales diseñados por Apple)

🔹 **Ejemplos de casos de uso:**

- Pedir comida sin descargar la aplicación de un restaurante
- Pagar por el estacionamiento
- Alquiler de bicicletas o scooters
- Registrarse en un hotel

¿Cómo implementar App Clips?

1. **Crear un destino de App Clip en Xcode**
   - Abre tu proyecto principal de Xcode.
   - Vaya a Archivo > nuevo destino >.
   - Selecciona App Clip y agrégalo a tu aplicación existente.

Esto crea una versión ligera de tu app con un identificador de paquete independiente.

2. **Diseñe una interfaz de usuario y funcionalidad livianas**
   - Manténgalo por debajo de 10 MB.
   - Cargue solo activos esenciales.
   - Utilice recursos bajo demanda para activos adicionales si es necesario.
   - Limita el propósito del clip de la app a una sola tarea.

3. **Implementación de la invocación de App Clip mediante NSUserActivity**

En SceneDelegate.swift del App Clip, controla la URL entrante:

```
func scene(_ scene: UIScene, continue userActivity: NSUserActivity) {
        guard let incomingURL = userActivity.webpageURL else { return }

        handleIncomingURL(incomingURL)
}
```

Esto permite que los App Clips reconozcan cuando se inician a través de una URL o etiqueta NFC.

4. **Configurar dominios asociados**
    - En Signing & Capabilities, agregue la capacidad Associated Domains.
    - Incluye el dominio de tu sitio web applinks: y con los prefijos appclip:
        - applinks: yourdomain.com
        - appclip: yourdomain.com
    - Implementa un archivo de asociación de sitios de aplicaciones de Apple (AASA) en https://yourdomain.com/.well-known/apple-app-site-association con App Clip details.

Ejemplo de archivo AASA:

```
{
        "applinks": {
                "apps": [],
                "details": [{
                        "appID": "TEAM_ID.com.example.yourApp",
                        "paths": ["/order/*"]
                }]
        },

        "appclips": {
                "apps": ["TEAM_ID.com.example.yourAppClip"]
        }
}
```

5. **Agregar App Clip Experience en App Store Connect**
    - Ve a App Store Connect > tu aplicación.
    - Ve a Experiencias de App Clip.
    - Configure métodos de invocación (QR, NFC, URL).
    - Asócialo a una App Clip Card relevante (aparece en Safari, Mensajes, etc.).

6. **Prueba tu App Clip**

- Utilice la función de invocación de App Clip de Xcode.
- Escanee el Código QR de App Clip en el menú del desarrollador.
- Pruebe en Safari usando apple-app-site-association.

41. **¿Qué herramientas utiliza para perfilar y optimizar una aplicación iOS?**

La creación de perfiles y la optimización en iOS son cruciales para mantener el rendimiento. Las principales herramientas utilizadas son:

- **Xcode Instruments:** Ayuda a analizar fugas de memoria, uso de CPU, escrituras de disco, rendimiento de red, etc.
- **Time Profiler:** Detecta cuellos de botella en el uso de la CPU y el tiempo de ejecución de la función.

- **Leaks Instrument:** Identifica las fugas de memoria causadas por objetos retenidos.
- **Zombies Instruments:** Ayuda a depurar problemas de acceso a la memoria mediante la detección de mensajes enviados a objetos desasignados.
- **Registro de energía:** Monitorea el impacto energético, asegurando la eficiencia de la aplicación.
- **Generador de perfiles de red:** Supervisa los tiempos de respuesta de las llamadas a la API y el uso de la red.
- **Malloc y VM Tracker:** Analiza los patrones de asignación y desasignación de memoria.
- **Static Analyzer:** Detecta problemas en el código antes del tiempo de ejecución.

42. ¿Cómo se reduce la huella de memoria de una aplicación iOS?

La reducción de la huella de memoria garantiza un mejor rendimiento y estabilidad. Algunas de las mejores prácticas incluyen:

- **Utiliza ARC (recuento automático de referencias)** para administrar la memoria automáticamente.
- **Evita los ciclos de retención** mediante el uso de referencias débiles o sin propietario en los cierres y los patrones de delegación.
- **Manejo eficiente de imágenes:** Usa UIImage(named:) en lugar de UIImage(contentsOfFile:) y preferiblemente ImageAssets.
- **Usa estructuras de datos eficientes en memoria** como Set y Dictionary en lugar de matrices cuando corresponda.
- **Desecha los objetos no utilizados** estableciendo las variables en nil cuando ya no sean necesarias.
- **Evita el uso excesivo de singletons**, ya que persisten en la memoria durante todo el ciclo de vida de la aplicación.
- **Comprime imágenes y use el formato WebP** para una mejor eficiencia.
- **Libera los objetos y cachés no utilizados** durante las advertencias de memoria.

43. ¿Cómo ayuda Instruments a detectar fugas de memoria?

Instruments proporciona un análisis detallado de la memoria con estas características clave:

- **Leaks Instrument:** detecta fugas de memoria mediante el seguimiento de objetos no referenciados que aún existen en la memoria.
- **Allocations Instrument:** Supervisa el uso de la memoria y resalta la asignación excesiva.
- **Zombies Instrument:** Detecta los mensajes enviados a objetos desasignados, evitando accidentes.
- **Análisis de montones:** captura estados de memoria en diferentes puntos de ejecución, identificando objetos que persisten inesperadamente.
- **Detección de ciclos de retención:** busca objetos que tienen referencias fuertes entre sí, lo que evita la desasignación.

Con estas herramientas, los desarrolladores pueden identificar y corregir las fugas de memoria, evitando la degradación del rendimiento.

44. Explica Lazy loading y sus beneficios en una aplicación iOS.

Lazy loading es una técnica en la que los datos o recursos se cargan solo cuando es necesario en lugar de durante el inicio de la aplicación.

Beneficios:

- **Reduce el consumo inicial de memoria**, mejorando la velocidad de inicio de la aplicación.
- **Optimiza la utilización de recursos**, cargando activos pesados solo cuando es necesario.
- **Mejora el rendimiento del desplazamiento** en listas (por ejemplo, UITableView y UICollectionView) cargando imágenes de forma asíncrona.
- **Mejora la duración de la batería** al reducir el procesamiento innecesario.

Ejemplo: Lazy loading en una imagen en UITableViewCell:

```
class CustomCell: UITableViewCell {
        var imageURL: URL? {
                didSet {
                        loadImage()
                }
        }

        private func loadImage() {
                guard let url = imageURL else { return }

                DispatchQueue.global().async {
                        if let data = try? Data(contentsOf: url),
                        let image = UIImage(data:data) {
                                DispatchQueue.main.async {
                                        self.imageView?. image = image
                                }
                        }
                }
        }
}
```

45. ¿Cuáles son las prácticas recomendadas para manejar imágenes grandes en una aplicación iOS?

- Utiliza ImageAssets y gráficos vectoriales: preferiblemente símbolos SF o PDFs sobre imágenes rasterizadas de alta resolución.
- Usar formato WebP: Proporciona una mejor compresión que PNG o JPEG.
- Cambiar el tamaño de las imágenes antes de mostrarlas: Evite almacenar imágenes de gran tamaño en la memoria.
- Cargar imágenes de forma diferida: use bibliotecas como SDWebImage o Kingfisher para la carga y el almacenamiento en caché asíncronos.
- Usa Core Animation & Metal: Renderiza imágenes de manera eficiente sin bloquear el hilo principal.
- Evita usar UIImage(contentsOfFile:) usa UIImage(named:) para obtener ventajas de almacenamiento en caché.

46. ¿Cómo se optimiza el rendimiento de Core Data para aplicaciones a gran escala?

- Usa BackgroundContexts: Realiza operaciones pesadas en subprocesos en segundo plano.
- Inserciones por lotes y actualizaciones: Minimiza las operaciones de escritura mediante NSBatchInsertRequest.
- Indexación: Indexa los atributos consultados con frecuencia para acelerar las solicitudes de recuperación.
- Fallas y carga diferida: Evita precargar todos los objetos en la memoria.
- Compactar la base de datos SQLite: Ejecutar NSPersistentStoreCoordinator.execute(_:) periódicamente.
- Uso de FetchLimits & Predicates: Evita recuperar grandes conjuntos de datos innecesariamente.

47. ¿Cuáles son algunas estrategias para mejorar el tiempo de inicio de la aplicación?

- Reducir el tamaño del paquete de aplicaciones: elimine los activos innecesarios y use la reducción de aplicaciones.
- Aplazar el trabajo no esencial: cargue solo los componentes críticos al inicio.
- Optimizar guiones gráficos: Evite usar un solo guión gráfico masivo.
- Objetos pesados de carga diferida: aplaza la carga de la base de datos y las llamadas de red.

- Precargar datos de manera eficiente: utilice mecanismos de recuperación y almacenamiento en caché en segundo plano.
- Optimización de inicializadores estáticos: evite cálculos costosos en AppDelegate.

48. Explicar cómo controlar la ejecución en segundo plano y la administración del ciclo de vida de la aplicación.

iOS proporciona modos de ejecución en segundo plano:

- **Búsqueda en segundo plano:** Obtiene nuevos datos periódicamente (setMinimumBackgroundFetchInterval).
- **Notificaciones push silenciosas:** Actualiza el contenido de la aplicación sin interacción del usuario.
- **Tareas en segundo plano (BGTaskScheduler):** Ejecuta tareas de larga duración en segundo plano.
- **Audio, ubicación y servicios de VoIP:** Casos especiales en los que las aplicaciones continúan ejecutándose.

Gestión del ciclo de vida:

- applicationDidEnterBackground: Guarda datos y libera recursos.
- applicationWillEnterForeground: Restaura el estado y actualiza la interfaz de usuario.
- applicationDidBecomeActive: Reinicia las tareas en pausa.
- applicationWillTerminate: Realiza la limpieza final.

49. ¿Cuáles son algunas de las mejores prácticas de seguridad para almacenar datos confidenciales en una aplicación iOS?

- Usar servicios de llavero: Almacena credenciales de usuario confidenciales de forma segura.
- Cifrar datos locales: Utilice AES-256 para el cifrado de archivos locales.
- Evite los secretos codificados: nunca almacene claves o credenciales de API en el paquete de aplicación.
- Utilice enclaves seguros: almacene datos de autenticación biométrica de forma segura.
- Habilitar App TransportSecurity(ATS): Forzar conexiones HTTPS.
- Usar almacenamiento seguro: prefiera UserDefaults solo para datos no confidenciales.
- Implemente la autenticación de dos factores (2FA): para una mejor seguridad.

50. ¿Cómo implementaría el cifrado de extremo a extremo en una aplicación iOS?

El cifrado de extremo a extremo (E2EE) garantiza que los datos permanezcan cifrados desde el remitente hasta el receptor.

Pasos para implementar:

1. Generar claves de cifrado: Utilice el cifrado RSA (asimétrico) o AES (simétrico).

Cifre los datos antes de enviarlos:

```
let clave = SymmetricKey (tamaño: .bits256)
let sealedBox = pruebe AES. GCM.seal(data, using: key)
let encryptedData = sealedBox.combined
```

2. Transmitir de forma segura: usa HTTPS + TLS con anclaje de certificados.

Descifrar en el extremo del receptor:

```
let sealedBox = pruebe AES. GCM. SealedBox(combinado: encryptedData)
let decryptedData = pruebe AES. GCM.open(sealedBox, usando: clave)
```

3. Utiliza el intercambio seguro de claves: implemente Diffie-Hellman o criptografía de curva elíptica (ECC).

4. Utiliza Secure Enclave de Apple para la administración de claves: almacene claves privadas de forma segura.