

# SwiftUI Moderne & Liquid Glass

---

## Design iOS 26 & Performance

---

### Jour 2 - Atelier Swift Avancé

# Au menu du chef

Partie 1 (40min) : Cycle de Vie SwiftUI

Partie 2 (50min) : Patterns de Performance

Partie 3 (45min) : Animations & Transitions

Partie 4 (55min) : Liquid Glass iOS 26

Après-midi : TP - Implémentation de votre UI

16h00 / 16h30 : Fireside Chat avec un special guest

# Partie I



## Cycle de Vie SwiftUI

# Comment SwiftUI Fonctionne

**View Struct → body → View Tree → UIView/NSView**



## Concepts clés :

- Views = descriptions, pas des objets
- Body recalculé à chaque changement
- Diffing intelligent du View Tree
- Updates minimaux du vrai UI

# View Identity & Lifecycle

```
struct ContentView: View {  
    @State private var count = 0  
  
    var body: some View {  
        VStack {  
            Text("Count: \(count)")  
                .id(count) // Force nouvelle identity  
  
            Button("Increment") {  
                count += 1  
            }  
        }  
    }  
}
```

Identity détermine :

- Si une View est "la même" entre updates
- Quand l'état est préservé ou reset
- Ordre des animations

# @State, le property wrapper fondamental

```
struct CounterView: View {  
    @State private var count = 0  
  
    var body: some View {  
        Button("Count: \(count)") {  
            count += 1 // Déclenche re-render  
        }  
    }  
}
```

## Règles :

- **@State** : source de vérité locale à la View
- **private** toujours (encapsulation)
- Value types uniquement (Int, String, struct)
- SwiftUI gère le storage

# @Binding pour communication parent-enfant

```
struct ParentView: View {  
    @State private var isOn = false  
  
    var body: some View {  
        ToggleView(isOn: $isOn) // $ = Binding  
    }  
}  
  
struct ToggleView: View {  
    @Binding var isOn: Bool  
  
    var body: some View {  
        Toggle("Switch", isOn: $isOn)  
    }  
}
```

**@Binding** : référence bidirectionnelle à un **@State**

- Lecture ET écriture
- Pas de stockage (pointe vers **@State** parent)

# Property Wrappers : @Observable (iOS 17+)

```
import Observation

@Observable
class ViewModel {
    var name = ""
    var age = 0

    func updateName(_ newName: String) {
        name = newName
    }
}

struct ContentView: View {
    @State var viewModel = ViewModel()

    var body: some View {
        TextField("Name", text: $viewModel.name)
    }
}
```

# @Observable vs ObservableObject

```
// Ancien (iOS 13–16)
class OldViewModel: ObservableObject {
    @Published var name = ""
    @Published var age = 0
}
// Usage : @StateObject, @ObservedObject

// Nouveau (iOS 17+)
@Observable
class NewViewModel {
    var name = ""
    var age = 0
}
// Usage : @State, @Bindable
```

## Avantages

### @Observable :

- Syntaxe plus simple
- Performance améliorée (tracking granulaire)
- Marche avec optionals et collections

# @Environment pour valeurs injectées dans la hiérarchie

```
struct ContentView: View {
    @Environment(\.colorScheme) var colorScheme
    @Environment(\.dismiss) var dismiss

    var body: some View {
        Text("Mode: \(colorScheme == .dark ? "Dark" : "Light")")
        Button("Close") {
            dismiss()
        }
    }
}

// Custom environment
private struct ThemeKey: EnvironmentKey {
    static let defaultValue = Theme.default
}

extension EnvironmentValues {
    var theme: Theme {
        get { self[ThemeKey.self] }
        set { self[ThemeKey.self] = newValue }
    }
}
```

⚠️ Les Environment Values se propagent vers le bas de la hiérarchie uniquement

## Custom Environment Values

```
// 1. Définir une EnvironmentKey
private struct ThemeKey: EnvironmentKey {
    static let defaultValue = AppTheme.default
}

// 2. Étendre EnvironmentValues
extension EnvironmentValues {
    var appTheme: AppTheme {
        get { self[ThemeKey.self] }
        set { self[ThemeKey.self] = newValue }
    }
}

// 3. Injecter dans la hiérarchie
ContentView()
    .environment(\.appTheme, .premium)

// 4. Utiliser dans n'importe quelle View
struct MyView: View {
    @Environment(\.appTheme) var theme

    var body: some View {
        Text("Hello").foregroundStyle(theme.primaryColor)
    }
}
```

# View Updates : Dependencies<sup>1</sup>

```
struct ProfileView: View {  
    let user: User  
    @State private var isExpanded = false  
  
    var body: some View {  
        VStack {  
            Text(user.name) // Dépend de user  
            if isExpanded { // Dépend de isExpanded  
                Text(user.bio)  
            }  
        }  
    }  
}
```

- Dependency tracking :**
- SwiftUI détecte automatiquement les dépendances
  - View se re-render si une dépendance change
  - Parties non-affectées ne se recalculent pas

---

<sup>1</sup> WWDC23 "Demystify SwiftUI performance"

# Partie 2



## Patterns de Performance

# Métriques de performance fondamentales

Les 3 métriques clés :

1. **Hangs** : Délai dans l'apparition d'une View
2. **Hitches** : Frames perdus pendant animation/scroll
3. **Memory** : Utilisation mémoire excessive

## Objectif :

- 60 fps = 16.7ms par frame
- 120 fps (ProMotion) = 8.3ms par frame

## Outils :

- Instruments (Time Profiler, SwiftUI)
- Xcode Debug Navigator
- Metal System Trace

# Anti-Pattern : Expensive Body

```
// ✗ MAUVAIS - Calcul lourd dans body
struct SlowView: View {
    @State private var data: [Item]

    var body: some View {
        let processedData = data
            .filter { $0.isActive }
            .sorted { $0.name < $1.name }
            .map { processItem($0) } // Lourd !

        List(processedData) { item in
            Text(item.name)
        }
    }
}
```

**Problème** : Ce calcul se refait à CHAQUE re-render !

# Solution : Computed Properties

```
// ✓ BON - Calcul en computed property
struct FastView: View {
    @State private var data: [Item]

    private var processedData: [ProcessedItem] {
        data
            .filter { $0.isActive }
            .sorted { $0.name < $1.name }
            .map { processItem($0) }
    }

    var body: some View {
        List(processedData) { item in
            Text(item.name)
        }
    }
}
```

**Avantage :** Même résultat, mais SwiftUI peut mieux optimiser.

# Pattern : ViewModels avec @Observable

```
@Observable
class ListViewModel {
    private(set) var processedData: [ProcessedItem] = []

    var rawData: [Item] = [] {
        didSet {
            updateProcessedData()
        }
    }

    private func updateProcessedData() {
        processedData = rawData
            .filter { $0.isActive }
            .sorted { $0.name < $1.name }
            .map { processItem($0) }
    }
}

struct FastView: View {
    @State private var viewModel = ListViewModel()

    var body: some View {
        List(viewModel.processedData) { item in
            Text(item.name)
        }
    }
}
```

# LazyVStack vs VStack

Règle : Utilisez Lazy\* pour listes longues (>50 items)

```
// VStack - Tous les enfants créés immédiatement
VStack {
    ForEach(1...1000, id: \.self) { i in
        HeavyView(number: i) // 1000 Views créées !
    }
}

// LazyVStack - Crées seulement quand visibles
ScrollView {
    LazyVStack {
        ForEach(1...1000, id: \.self) { i in
            HeavyView(number: i) // Seulement ~20 à la fois
        }
    }
}
```

# Identifiable & id()

```
struct Item: Identifiable {
    let id: UUID
    var name: String
}

// ✅ BON - id stable
List(items) { item in
    Text(item.name)
}

// ❌ MAUVAIS - id instable
List(items) { item in
    Text(item.name)
        .id(UUID()) // Nouveau UUID à chaque render !
}
```

**Règle d'or : L'id doit être stable et unique**

- Stable = ne change pas entre renders
- Unique = différent pour chaque item

# Pattern : Extracting Subviews

```
// ✗ MAUVAIS - Vue monolithique
struct HeavyListView: View {
    @State private var items: [Item]

    var body: some View {
        List(items) { item in
            HStack {
                AsyncImage(url: item.imageURL)
                    .frame(width: 50, height: 50)
                VStack(alignment: .leading) {
                    Text(item.name).font(.headline)
                    Text(item.description).font(.caption)
                }
                Spacer()
                Button("Details") { /* ... */ }
            }
        }
    }
}
```

# Pattern : Extracted Subviews

```
// ✓ BON - Subviews extraites
struct OptimizedListView: View {
    @State private var items: [Item]

    var body: some View {
        List(items) { item in
            ItemRow(item: item)
        }
    }
}

struct ItemRow: View {
    let item: Item

    var body: some View {
        HStack {
            AsyncImage(url: item.imageUrl)
                .frame(width: 50, height: 50)
            VStack(alignment: .leading) {
                Text(item.name).font(.headline)
                Text(item.description).font(.caption)
            }
            Spacer()
            Button("Details") { /* ... */ }
        }
    }
}
```

# @ViewBuilder : Custom Containers

```
struct Card<Content: View>: View {  
    let title: String  
    @ViewBuilder let content: () -> Content  
  
    var body: some View {  
        VStack(alignment: .leading) {  
            Text(title)  
                .font(.headline)  
            Divider()  
            content()  
        }  
        .padding()  
        .background(.background)  
        .cornerRadius(12)  
    }  
}
```

```
Card(title: "Profile") {  
    Text("Name: John")  
    Text("Age: 30")  
    Button("Edit") {}  
}
```

# Partie 3

---

## Animations & Transitions

# Animation Basics

```
struct AnimatedView: View {  
    @State private var isExpanded = false  
  
    var body: some View {  
        VStack {  
            Rectangle()  
                .fill(.blue)  
                .frame(width: isExpanded ? 300 : 100,  
                       height: isExpanded ? 300 : 100)  
            .animation(.spring(response: 0.6), value: isExpanded)  
  
            Button("Toggle") {  
                isExpanded.toggle()  
            }  
        }  
    }  
}
```

Deux approches :

1. **.animation()** modifier
2. **withAnimation { }** closure

# Animation Curves

```
// Linear  
.animation(.linear(duration: 1.0))  
  
// Ease in/out  
.animation(.easeInOut(duration: 0.5))  
  
// Spring (physique réaliste)  
.animation(.spring(response: 0.6, dampingFraction: 0.7))  
  
// Custom  
.animation(.timingCurve(0.2, 0.8, 0.2, 1.0, duration: 0.5))  
  
// Interpolating Spring (iOS 17+)  
.animation(.smooth(duration: 0.5))
```

**Spring recommandé**  
pour la plupart des cas

- Feels naturel
- Self-terminating (pas de durée fixe)
- Response = "bounciness", damping = "friction"

# Implicit vs Explicit Animations

```
struct AnimationDemo: View {  
    @State private var scale: CGFloat = 1.0  
  
    var body: some View {  
        VStack {  
            // Implicit - animation attachée à la View  
            Circle()  
                .scaleEffect(scale)  
                .animation(.spring(), value: scale)  
  
            // Explicit - animation dans l'action  
            Button("Animate") {  
                withAnimation(.spring()) {  
                    scale = scale == 1.0 ? 1.5 : 1.0  
                }  
            }  
        }  
    }  
}
```

## Différence :

- Implicit : anime automatiquement tout changement de value
- Explicit : contrôle fin de QUAND animer

# Transitions

```
struct TransitionDemo: View {  
    @State private var show = false  
  
    var body: some View {  
        VStack {  
            if show {  
                Text("Hello!")  
                    .transition(.scale.combined(with: .opacity))  
            }  
  
            Button("Toggle") {  
                withAnimation {  
                    show.toggle()  
                }  
            }  
        }  
    }  
}
```

## Transitions disponibles :

- **.opacity, .scale, .slide, .move(edge:)**
- **.asymmetric(insertion: removal:)**
- Combiner avec **.combined(with:)**

# Custom Transitions

```
extension AnyTransition {
    static var slideAndFade: AnyTransition {
        .asymmetric(
            insertion: .move(edge: .trailing)
                .combined(with: .opacity),
            removal: .move(edge: .leading)
                .combined(with: .opacity)
        )
    }
}

// Usage
if show {
    DetailView()
        .transition(.slideAndFade)
}
```

# Matched Geometry Effect

```
struct MatchedGeometryDemo: View {
    @State private var showDetail = false
    @Namespace private var animation

    var body: some View {
        if !showDetail {
            Circle()
                .fill(.blue)
                .frame(width: 100, height: 100)
                .matchedGeometryEffect(id: "circle", in: animation)
                .onTapGesture {
                    withAnimation(.spring()) {
                        showDetail = true
                    }
                }
        } else {
            Circle()
                .fill(.blue)
                .frame(width: 300, height: 300)
                .matchedGeometryEffect(id: "circle", in: animation)
                .onTapGesture {
                    withAnimation(.spring()) {
                        showDetail = false
                    }
                }
        }
    }
}
```

# Phase Animator (iOS 17+)

```
struct PhaseAnimationDemo: View {  
    var body: some View {  
        PhaseAnimator([false, true]) { phase in  
            Circle()  
                .fill(.blue)  
                .scaleEffect(phase ? 1.5 : 1.0)  
                .opacity(phase ? 0.5 : 1.0)  
        } animation: { phase in  
            .easeInOut(duration: 1.0)  
        }  
    }  
}
```

**PhaseAnimator** = séquence d'états animés

- Boucle automatiquement à travers les phases
- Différentes animations par phase
- Parfait pour loading, pulsing, etc.

# Keyframe Animator (iOS 17+)

```
struct KeyframeDemo: View {
    @State private var trigger = false

    var body: some View {
        Circle()
            .fill(.blue)
            .frame(width: 100, height: 100)
            .keyframeAnimator(
                initialValue: AnimationValues(),
                trigger: trigger
            ) { content, value in
                content
                    .scaleEffect(value.scale)
                    .rotationEffect(value.rotation)
            } keyframes: { _ in
                KeyframeTrack(\.scale) {
                    SpringKeyframe(1.5, duration: 0.3)
                    SpringKeyframe(1.0, duration: 0.3)
                }
                KeyframeTrack(\.rotation) {
                    LinearKeyframe(.degrees(45), duration: 0.3)
                    LinearKeyframe(.degrees(0), duration: 0.3)
                }
            }
    }
}

struct AnimationValues {
    var scale = 1.0
    var rotation = Angle.zero
}
```

# Partie 4



## Liquid Glass (iOS 26+)

# Qu'est-ce que Liquid Glass ?

Un nouveau design system unifié :

- iOS 26, iPadOS 26, macOS Tahoe 26
- watchOS 26, (~ tvOS 26, visionOS 2+)

Principe : "Digital meta-material" qui :

- Bend la lumière dynamiquement (lensing)
- Répond au touch avec élasticité
- S'adapte au contenu sous-jacent
- Crée de la profondeur sans obscurcir

Pas juste un blur : Optical effects réalistes

- Refraction aux edges
- Highlights qui bougent avec le device
- Shadows adaptatives selon le fond

# Liquid Glass : Regular vs Clear

Two Variants / Regular (défaut) :

- Adapte automatiquement light/dark
- Fonctionne à toute taille
- Effets visuels complets
- Use case : 90% des cas

⚠ Ne jamais mélanger les deux dans une app

# Liquid Glass : Regular vs Clear

## Two Variants / Clear :

- Transparence permanente élevée
- Nécessite dimming layer
- Pour contenus media-rich
- Use case : Photos, vidéos

⚠ Ne jamais mélanger les deux dans une app

## Regular (défaut) :

- Adapte automatiquement light/dark
- Fonctionne à toute taille
- Effets visuels complets
- Use case : 90% des cas

## Two Variants / Clear :

- Transparence permanente élevée
- Nécessite dimming layer
- Pour contenus media-rich
- Use case : Photos, vidéos

# glassEffect() Modifier

```
// Basic
Text("Hello, Glass!")
    .padding()
    .glassEffect()

// Avec variante
.glassEffect(.regular)
.glassEffect(.clear)

// Avec shape
.glassEffect(.regular, in: .capsule)
.glassEffect(.regular, in: .rect(cornerRadius: 20))

// Avec tint
.glassEffect(.regular.tint(.blue))

// Avec interactivité
.glassEffect(.regular.tint(.blue).interactive())
```

**Interactive** = répond au touch avec flexing

# GlassEffectContainer

```
import SwiftUI

GlassEffectContainer(spacing: 40.0) {
    HStack(spacing: 40.0) {
        Image(systemName: "pencil")
            .frame(width: 80, height: 80)
            .glassEffect()

        Image(systemName: "eraser")
            .frame(width: 80, height: 80)
            .glassEffect()

        Image(systemName: "paintbrush")
            .frame(width: 80, height: 80)
            .glassEffect()
    }
}
```

Pourquoi ? Glass ne peut pas échantillonner glass

- Container groupe les éléments
- Partage les ressources de sampling
- Permet morphing entre éléments

# Glass Morphing

```
struct MorphingDemo: View {
    @State private var showDetail = false
    @Namespace private var namespace

    var body: some View {
        if !showDetail {
            Button("Show More") {
                withAnimation(.smooth) {
                    showDetail = true
                }
            }
            .glassEffect()
            .glassEffectID("button", in: namespace)
        } else {
            VStack {
                Text("Details")
                Button("Hide") {
                    withAnimation(.smooth) {
                        showDetail = false
                    }
                }
            }
            .padding()
            .glassEffect()
            .glassEffectID("button", in: namespace)
        }
    }
}
```

# Button Styles Glass

```
// Glass button (secondary)
Button("Action") {
    // action
}
.buttonStyle(.glass)

// Glass Prominent (primary)
Button("Primary") {
    // action
}
.buttonStyle(.glassProminent)

// Avec tint
Button("Colored") {
    // action
}
.buttonStyle(.glass)
.tint(.blue)
```

Nouveaux styles :

- .glass = translucent, pour actions secondaires
- .glassProminent = opaque, pour primary actions

# NavigationSplitView avec Glass

```
NavigationSplitView {  
    List(items) { item in  
        NavigationLink(item.name, value: item)  
    }  
    .backgroundExtensionEffect()  
} detail: {  
    DetailView()  
}
```

## Nouveautés iOS 26 :

- Sidebar flotte au-dessus du content (glass)
- `.backgroundExtensionEffect()` étend le contenu sous la sidebar
- Content se mirror et blur en dehors du safe area

# TabView avec Glass<sup>2</sup>

```
TabView(selection: $selectedTab) {  
    HomeView()  
    .tabItem {  
        Label("Home", systemImage: "house")  
    }  
  
    SearchView()  
    .tabItem {  
        Label("Search", systemImage: "magnifyingglass")  
    }  
}  
.tabBarMinimizeBehavior(.onScrollDown)  
.tabViewBottomAccessory {  
    NowPlayingView()  
}
```

## Nouveautés :

- Tab bar flotte (glass)
- **.tabBarMinimizeBehavior()**  
cache sur scroll
- **.tabViewBottomAccessory()**  
pour contenu persistant

---

<sup>2</sup> WWDC25 "Build a SwiftUI app with the new design" montre tout ça en action avec l'app Landmarks.

# Sheets avec Glass

```
struct SheetDemo: View {
    @State private var showSheet = false

    var body: some View {
        Button("Show Sheet") {
            showSheet = true
        }
        .sheet(isPresented: $showSheet) {
            VStack {
                Text("Sheet Content")
                Spacer()
            }
            .presentationDetents([.medium, .large])
            // ⚠ Pas de .presentationBackground()
            // Le glass s'applique automatiquement
        }
    }
}
```

Changement : Sheets ont glass automatiquement

- Retirez `.presentationBackground()`
- Edges s'incurvent aux petits detents
- Morphing depuis button possible

# UIKit : UIGlassEffect

```
import UIKit

let effectView = UIVisualEffectView()
let glassEffect = UIGlassEffect()

// Configuration
glassEffect.isInteractive = true
glassEffect.tintColor = .systemBlue

// Application
UIView.animate(withDuration: 0.3) {
    effectView.effect = glassEffect
}

// Variants
let regularGlass = UIGlassEffect()
let clearGlass = UIGlassEffect.clear()
```

Support UIKit complet :

- **UIGlassEffect** pour l'effet
- **UIVisualEffectView** comme container
- Animation avec **UIView.animate**

# Design Principles : Où Utiliser Glass

Règle : Glass sépare navigation du contenu

Navigation Layer uniquement :

- Toolbars
- Tab bars
- Sidebars
- Sheets
- Menus
- Floating buttons

Pas sur Content Layer :

- Table views / Lists
- Cards de contenu
- Images
- Text areas

# Design Principles : Tinting

```
// ✓ BON - Tint avec signification
Button("Checkout") {
    // checkout action
}
.buttonStyle(.glassProminent)
.tint(.red) // Rouge = attention, action importante

// ✗ MAUVAIS - Tint décoratif partout
VStack {
    Button("Action 1").tint(.blue)
    Button("Action 2").tint(.green)
    Button("Action 3").tint(.purple)
    Button("Action 4").tint(.orange)
}
```

## Tinting rules :

- Avec signification, pas décoration
- Un ou deux tints max par écran
- Tints s'adaptent au background (light/dark)

# Performance : GPU Rendering

Liquid Glass utilise  
Metal shaders :

- Gaussian blur GPU-accéléré
- Fragment shaders en parallèle
- 60-120 fps maintenable

Device requirements :

- iOS 26 nécessite A13+ (iPhone 11+)
- A17 Pro / A18 : performance optimale
- iPhone 12-13 : peut avoir lag

Optimizations :

- Max 16 shapes par GlassEffectContainer
- Glass thickness < 20px recommandé
- Glass statique > glass mobile

# Performance : Battery Impact

## Tests réels :

- 8-12% CPU overhead vs interfaces statiques
- Impact batterie minimal sur A17+ / A18
- Dark mode + glass = meilleur sur OLED

## iOS 26.1 Toggle :

- Settings → Display & Brightness → Liquid Glass
- Clear (défaut, plus transparent)
- Tinted (plus opaque, meilleur contraste)

## Profiling tools :

- Metal System Trace (Instruments)
- Metal Debugger (Xcode)
- Target < 8.3ms par frame (120fps)

# Accessibility : Adaptation Automatique

```
@Environment(\.accessibilityReduceTransparency)  
var reduceTransparency  
  
if reduceTransparency {  
    // Alternative UI  
}
```

Settings respectés automatiquement :

- Reduce Transparency → glass plus opaque
- Increase Contrast → black/white avec borders
- Reduce Motion → moins d'élasticité

Developer responsibilities :

- Maintenir 4.5:1 contrast ratio (WCAG)
- Utiliser SF Symbols (VoiceOver OK)
- Labels texte en plus des icônes

# Migration : Automatic Adoption<sup>3</sup>

Apps SwiftUI iOS 18+ :

1. Recompile avec Xcode 16 (iOS 26 SDK)
2. C'est tout ! Glass appliqué automatiquement

Cleanup nécessaire :

```
swift
// Retirer ces modifiers obsolètes
.toolbarBackground(.visible, for: .navigationBar)
.presentationBackground(.ultraThinMaterial)
.background(.regularMaterial)
```

Opt-out temporaire (⚠⚠⚠⚠ expire iOS 27

⚠⚠⚠⚠) :

```
xml
<!-- Info.plist -->
<key>UIDesignRequiresCompatibility</key>
<true/>
```

---

<sup>3</sup> WWDC25 Session 208 "Showcase: Apps integrating Liquid Glass" avec retours d'expérience real-world de plusieurs grandes apps.

# Migration : Timeline Réelle

Safari team (Apple) :

- Estimation : 1.5 semaines
- Réalité : 1.5 jours
- Raison : Utilisaient déjà composants natifs

American Airlines :

- Migration rapide grâce à SwiftUI natif
- Pas de UIKit custom à migrer

Recommendation :

- Migrez progressivement
- Testez sur devices réels (pas que simulateur)
- Utilisez les bêta iOS !

# Custom Glass Effects

```
struct CustomGlassCard<Content: View>: View {
    @ViewBuilder let content: Content

    var body: some View {
        content
            .padding()
            .frame(maxWidth: .infinity)
            .glassEffect(.regular.tint(.blue).interactive())
            .cornerRadius(20)
            .shadow(color: .black.opacity(0.1),
                    radius: 10, y: 5)
    }
}

// Usage
CustomGlassCard {
    VStack(alignment: .leading) {
        Text("Title").font(.headline)
        Text("Content")
    }
}
```

# Liquid Glass : Best Practices

**DO :**

- Utiliser pour navigation uniquement
- GlassEffectContainer pour éléments proches
- Tint avec signification
- Tester sur devices réels
- Respecter Regular vs Clear

**DON'T :**

- Glass sur contenu
- Mélanger Regular et Clear
- Trop de tints
- Glass sur glass
- Oublier accessibility

# Ressources Officielles

## Documentation Apple :

- Liquid Glass Overview
- Adopting Liquid Glass
- Applying Liquid Glass to custom views
- Human Interface Guidelines

## WWDC 2025 Sessions :

- Session 219 : Meet Liquid Glass
- Session 323 : Build SwiftUI app with new design
- Session 284 : Build UIKit app with new design
- Session 356 : Get to know the new design system
- Session 208 : Apps integrating Liquid Glass

## Sample Code :

- Landmarks project (updated for iOS 26)

# Récapitulatif Jour 2

Ce qu'on a couvert :

**Cycle de Vie** : @State, @Binding, @Observable, @Environment

**Performance** : Body optimization, LazyVStack, Subviews

**Animations** : Springs, Transitions, Matched Geometry

**Liquid Glass** : glassEffect(), GlassEffectContainer, Regular vs Clear

Prochaine étape :

**14h-16h00** : TP

**16h-16h30** : Fireside Chat with...

**16h30-18h00** : TP

# Questions ?

## Liens Rapides

SwiftUI : [developer.apple.com/swiftui](https://developer.apple.com/swiftui)

Performance : WWDC23 Session 10160

Animations : WWDC23 "Animate with springs"

Liquid Glass : WWDC25 Session 219

# Fireside Chat

Dans 5 minutes

Guest : 

Sujet : All things 

Préparez vos questions !

