Seminář 1: Úvod do iOS vývoje

Tvorba mobilních aplikací

Roman Vyjídáček

Kdo jsem

- Software engineer ve společnosti Enverus
- Absolvent doktorského studia informatiky na UP
- Mám cca 8 let zkušenosti s vývojem pro iOS
- Předchozí projekty:
  - UPlikace
  - Noc vědců

Organizační informace

- Úkoly budu kontrolovat pouze na semináři
- Na vypracování máte čas do následujícího semináře
- Přístup k Macům bude pouze v měsíci duben
- Poslední úkoly budou zadány 23. dubna → 30. dubna je poslední šance na odevzdání úkolů
- Máme jen 6 seminářů → prezentovaná látka je jen přehledem, což znamená, že dokumentace bude Váš nejlepší přítel
- Konzultace po domluvě a ideálně online přes MS Teams

Obsah semináře

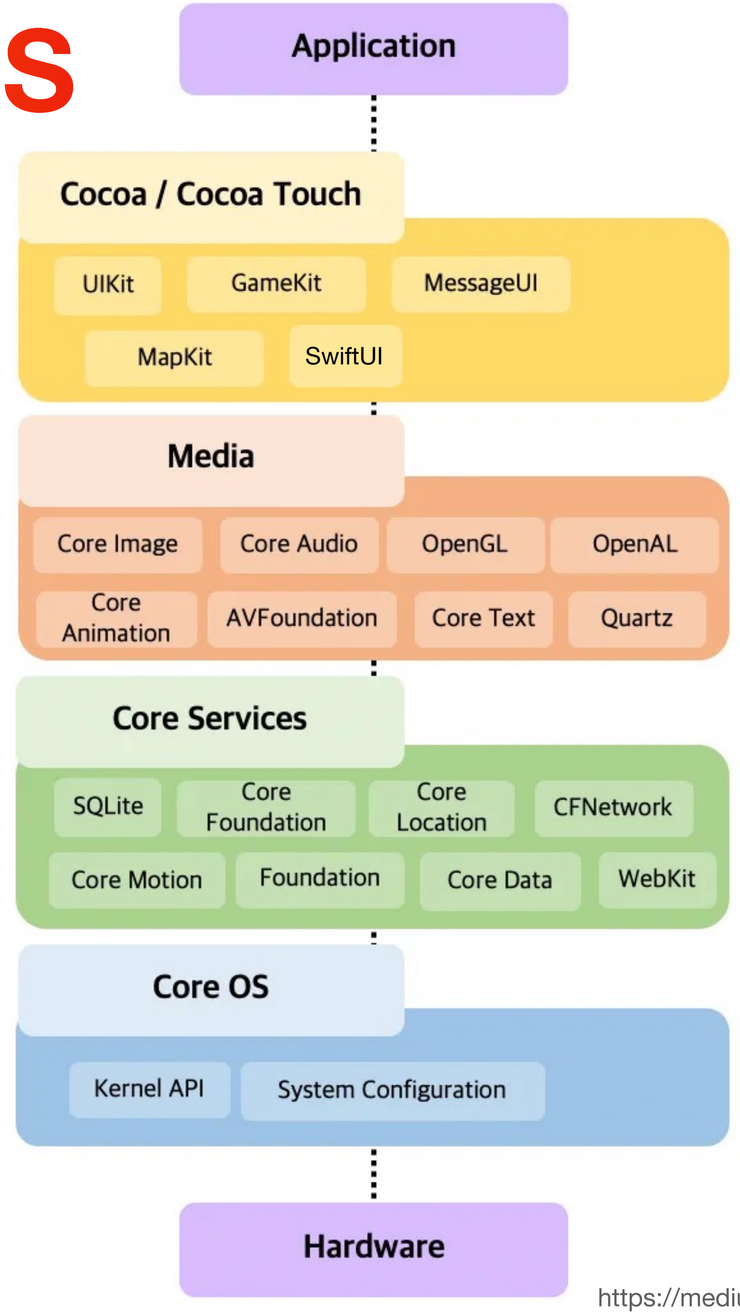
- Platforma iOS
- Vývojové prostředí
- Základy jazyka Swift

Platforma iOS

Operační systém

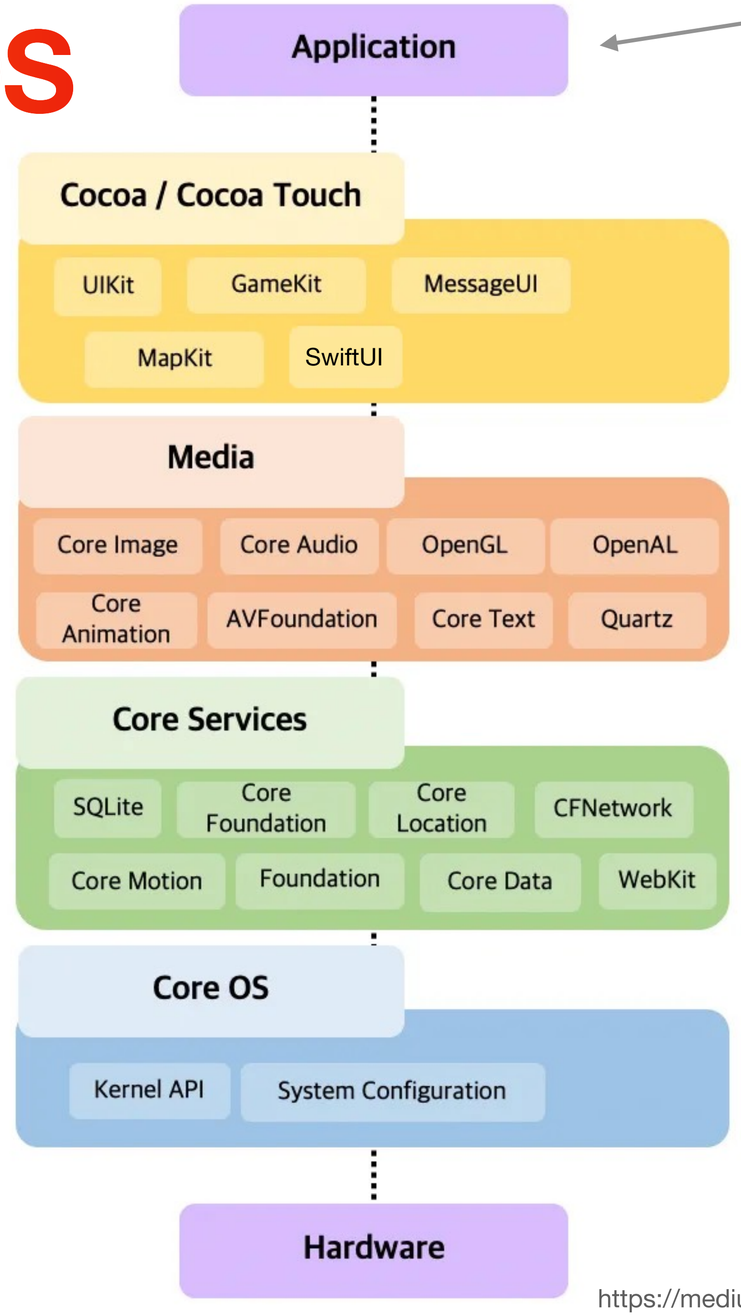
- Vychází z desktopového macOS.
- Uzavřený
- Jailbreak - obdoba root na android
- Podpora multitouch
- Multitasking vs. energetická náročnost

Architektura iOS



https://medium.com/@ganeshrajugalla/ios-ios-introduction-and-structure-fdd7ecf08c4c

Architektura iOS

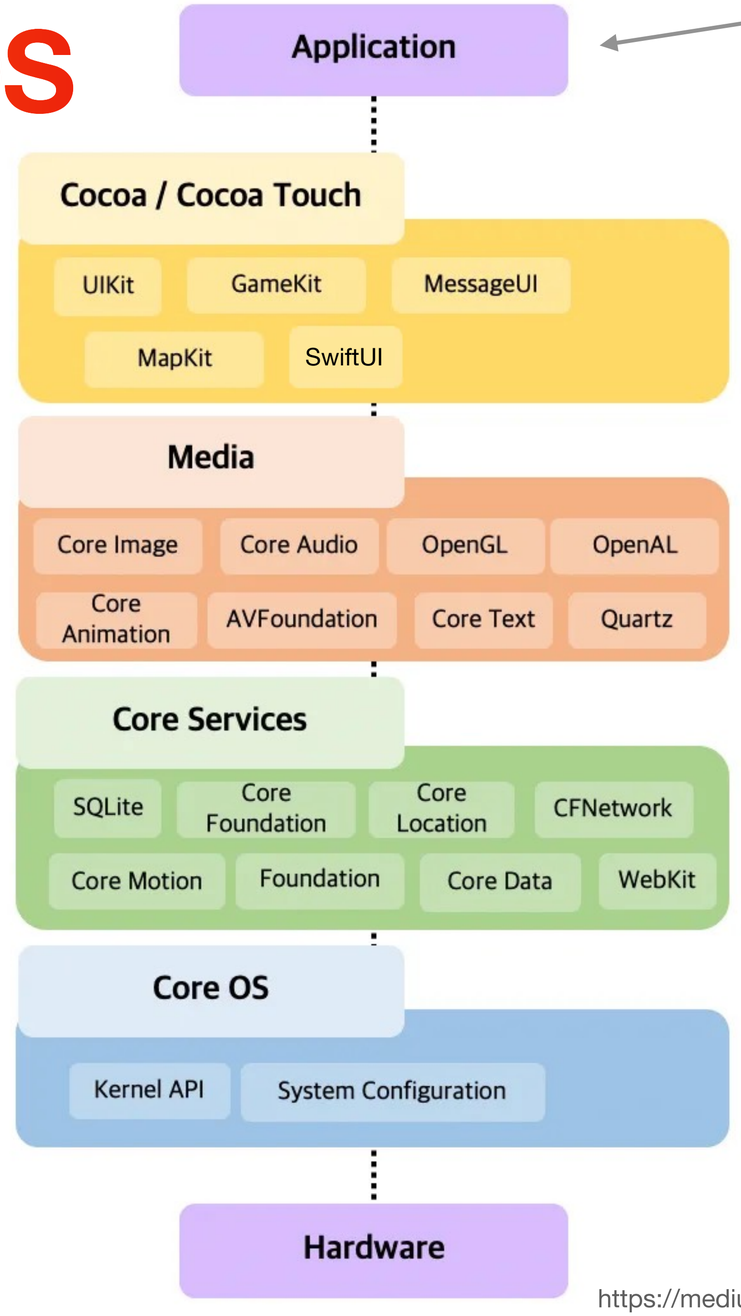


Application Layer (Vrstva aplikací)
Tato vrstva představuje samotné aplikace, které uživatelé spouštějí na svých zařízeních. Komunikuje s nižšími vrstvami pomocí systémových frameworků a poskytuje uživatelské rozhraní a funkčnost aplikací.

https://medium.com/@ganeshrajugalla/ios-ios-introduction-and-structure-fdd7ecf08c4c

Architektura iOS

Cocoa / Cocoa Touch
Framework pro vývoj iOS aplikací, který zajišťuje práci s uživatelským rozhraním a interakcemi. Obsahuje knihovny jako SwiftUI pro zobrazení prvků, GameKit pro herní funkce, MessageUI pro zprávy a MapKit pro mapy.

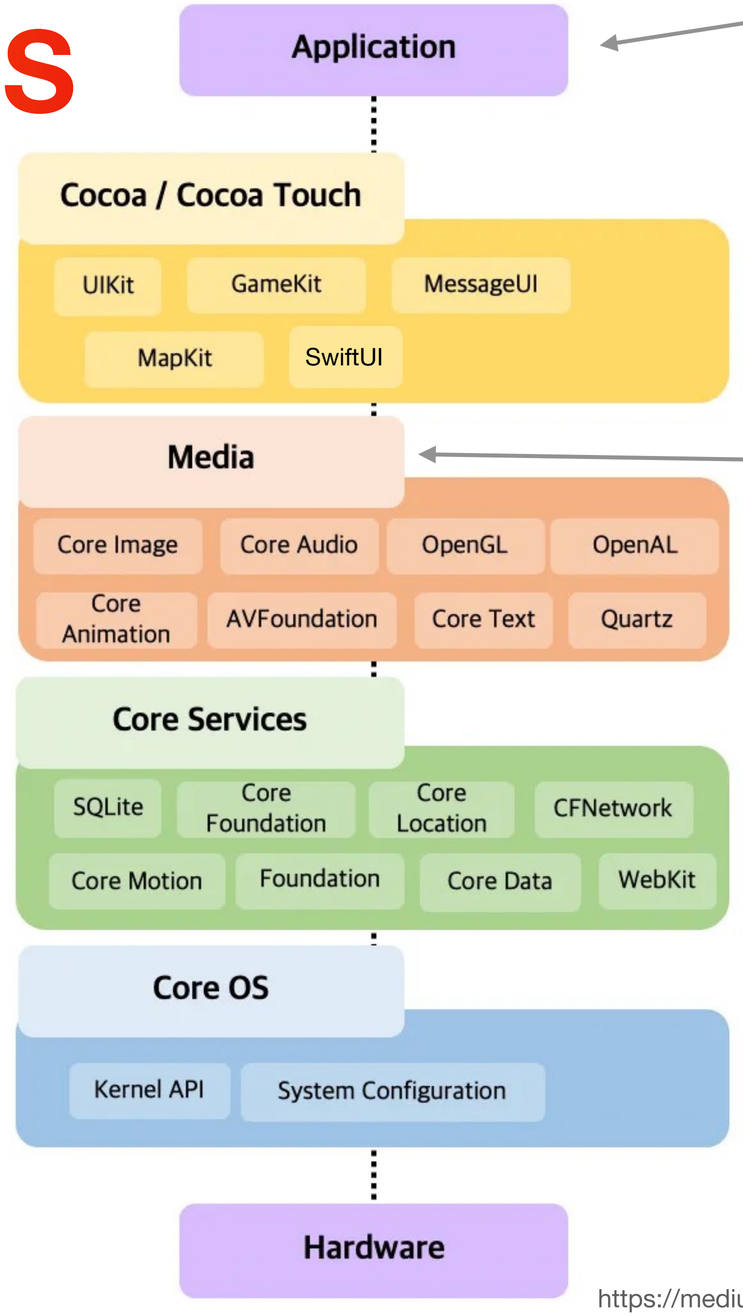


Application Layer (Vrstva aplikací)
Tato vrstva představuje samotné aplikace, které uživatelé spouštějí na svých zařízeních. Komunikuje s nižšími vrstvami pomocí systémových frameworků a poskytuje uživatelské rozhraní a funkčnost aplikací.

https://medium.com/@ganeshrajugalla/ios-ios-introduction-and-structure-fdd7ecf08c4c

Architektura iOS

Cocoa / Cocoa Touch
Framework pro vývoj iOS aplikací, který zajišťuje práci s uživatelským rozhraním a interakcemi. Obsahuje knihovny jako SwiftUI pro zobrazení prvků, GameKit pro herní funkce, MessageUI pro zprávy a MapKit pro mapy.



Application Layer (Vrstva aplikací)
Tato vrstva představuje samotné aplikace, které uživatelé spouštějí na svých zařízeních. Komunikuje s nižšími vrstvami pomocí systémových frameworků a poskytuje uživatelské rozhraní a funkčnost aplikací.

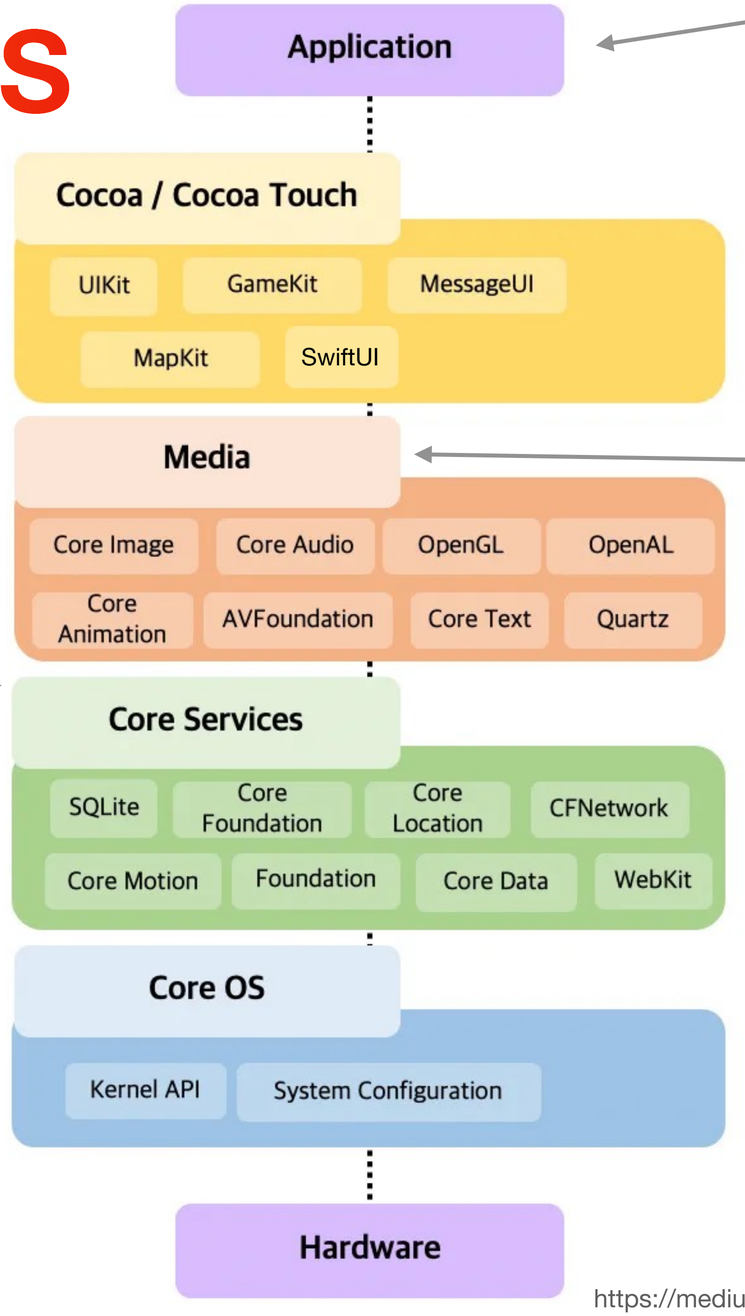
Media Layer
Vrstva zpracovávající multimédia – zvuk, video, grafiku a animace. Obsahuje technologie jako Core Image pro úpravy obrázků, Core Audio pro zvuk, OpenGL pro 3D grafiku a AVFoundation pro práci s videem.

https://medium.com/@ganeshrajugalla/ios-ios-introduction-and-structure-fdd7ecf08c4c

Architektura iOS

Cocoa / Cocoa Touch
Framework pro vývoj iOS aplikací, který zajišťuje práci s uživatelským rozhraním a interakcemi. Obsahuje knihovny jako SwiftUI pro zobrazení prvků, GameKit pro herní funkce, MessageUI pro zprávy a MapKit pro mapy.

Core Services
Zajišťuje klíčové služby pro aplikace, jako je správa databází (SQLite, Core Data), síťová komunikace (CFNetwork), GPS lokalizace (Core Location) a senzorická data (Core Motion). Tato vrstva umožňuje aplikacím efektivně pracovat s daty a hardwarem.



Application Layer (Vrstva aplikací)
Tato vrstva představuje samotné aplikace, které uživatelé spouštějí na svých zařízeních. Komunikuje s nižšími vrstvami pomocí systémových frameworků a poskytuje uživatelské rozhraní a funkčnost aplikací.

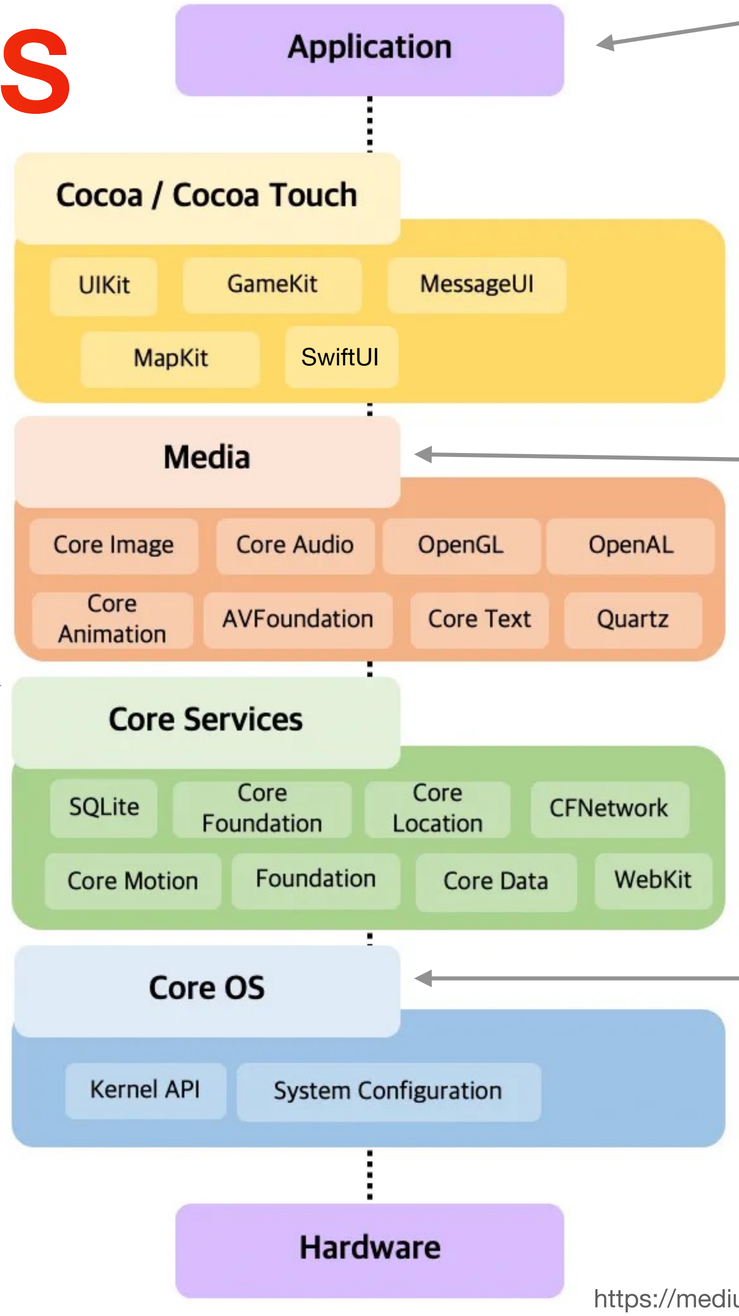
Media Layer
Vrstva zpracovávající multimédia – zvuk, video, grafiku a animace. Obsahuje technologie jako Core Image pro úpravy obrázků, Core Audio pro zvuk, OpenGL pro 3D grafiku a AVFoundation pro práci s videem.

https://medium.com/@ganeshrajugalla/ios-ios-introduction-and-structure-fdd7ecf08c4c

Architektura iOS

Application Layer (Vrstva aplikací)
Tato vrstva představuje samotné aplikace, které uživatelé spouštějí na svých zařízeních. Komunikuje s nižšími vrstvami pomocí systémových frameworků a poskytuje uživatelské rozhraní a funkčnost aplikací.

Cocoa / Cocoa Touch
Framework pro vývoj iOS aplikací, který zajišťuje práci s uživatelským rozhraním a interakcemi. Obsahuje knihovny jako SwiftUI pro zobrazení prvků, GameKit pro herní funkce, MessageUI pro zprávy a MapKit pro mapy.



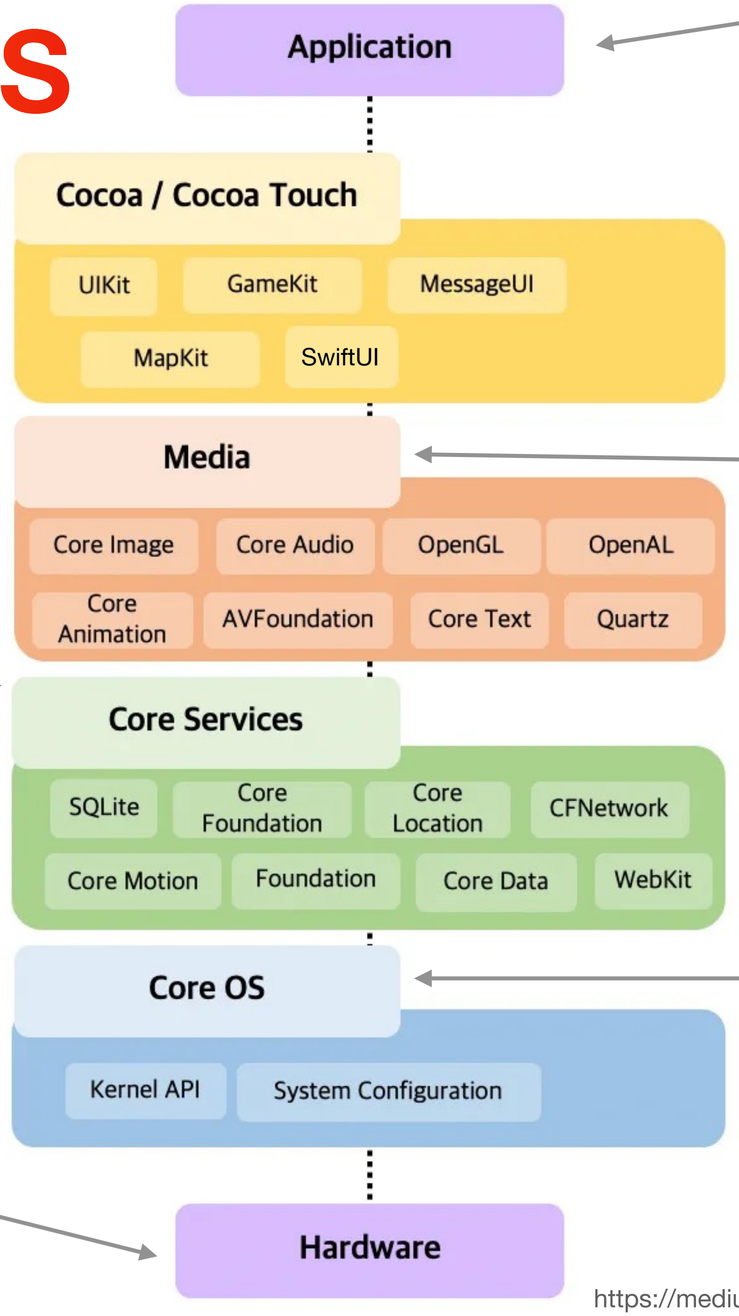
Media Layer
Vrstva zpracovávající multimédia – zvuk, video, grafiku a animace. Obsahuje technologie jako Core Image pro úpravy obrázků, Core Audio pro zvuk, OpenGL pro 3D grafiku a AVFoundation pro práci s videem.

Core Services
Zajišťuje klíčové služby pro aplikace, jako je správa databází (SQLite, Core Data), síťová komunikace (CFNetwork), GPS lokalizace (Core Location) a senzorická data (Core Motion). Tato vrstva umožňuje aplikacím efektivně pracovat s daty a hardwarem.

Core OS
Nejnižší softwarová vrstva, která komunikuje přímo s hardwarem. Obsahuje jádro operačního systému (Kernel API), které spravuje paměť, procesy a bezpečnost. System Configuration zajišťuje správu síťových nastavení a systémových oprávnění.

https://medium.com/@ganeshrajugalla/ios-ios-introduction-and-structure-fdd7ecf08c4c

Architektura iOS



Application Layer (Vrstva aplikací)
Tato vrstva představuje samotné aplikace, které uživatelé spouštějí na svých zařízeních. Komunikuje s nižšími vrstvami pomocí systémových frameworků a poskytuje uživatelské rozhraní a funkčnost aplikací.

Cocoa / Cocoa Touch
Framework pro vývoj iOS aplikací, který zajišťuje práci s uživatelským rozhraním a interakcemi. Obsahuje knihovny jako SwiftUI pro zobrazení prvků, GameKit pro herní funkce, MessageUI pro zprávy a MapKit pro mapy.

Media Layer
Vrstva zpracovávající multimédia – zvuk, video, grafiku a animace. Obsahuje technologie jako Core Image pro úpravy obrázků, Core Audio pro zvuk, OpenGL pro 3D grafiku a AVFoundation pro práci s videem.

Core Services
Zajišťuje klíčové služby pro aplikace, jako je správa databází (SQLite, Core Data), síťová komunikace (CFNetwork), GPS lokalizace (Core Location) a senzorická data (Core Motion). Tato vrstva umožňuje aplikacím efektivně pracovat s daty a hardwarem.

Core OS
Nejnižší softwarová vrstva, která komunikuje přímo s hardwarem. Obsahuje jádro operačního systému (Kernel API), které spravuje paměť, procesy a bezpečnost. System Configuration zajišťuje správu síťových nastavení a systémových oprávnění.

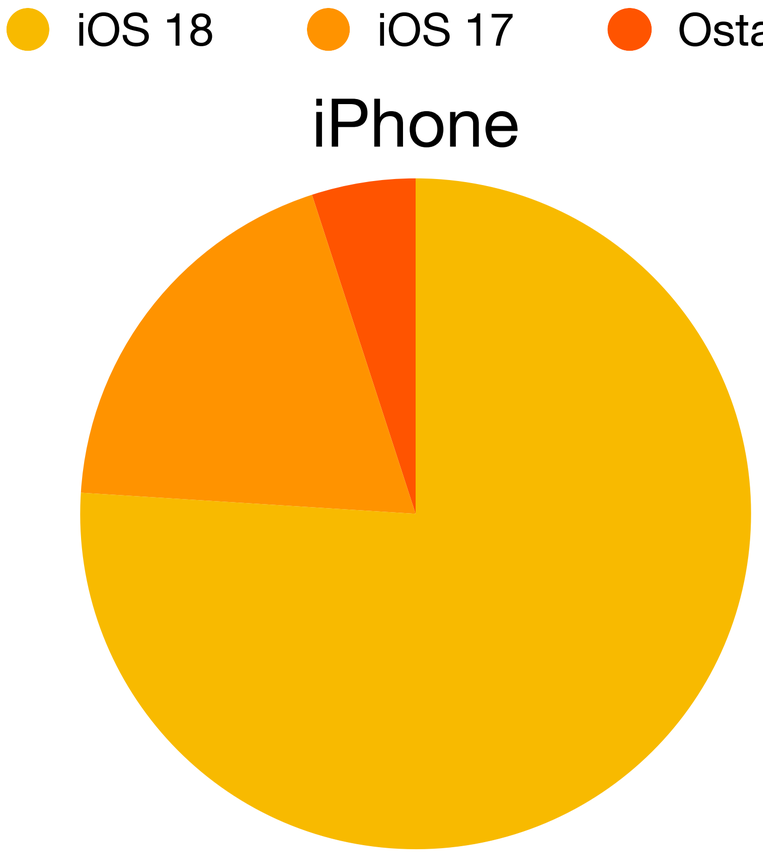
Hardware
Fyzická zařízení, na kterých běží iOS. Patří sem iPhone, iPad nebo Apple Watch. Tato vrstva spolupracuje s Core OS pro přímý přístup k procesoru, paměti, baterii a senzorům.

https://medium.com/@ganeshrajugalla/ios-ios-introduction-and-structure-fdd7ecf08c4c

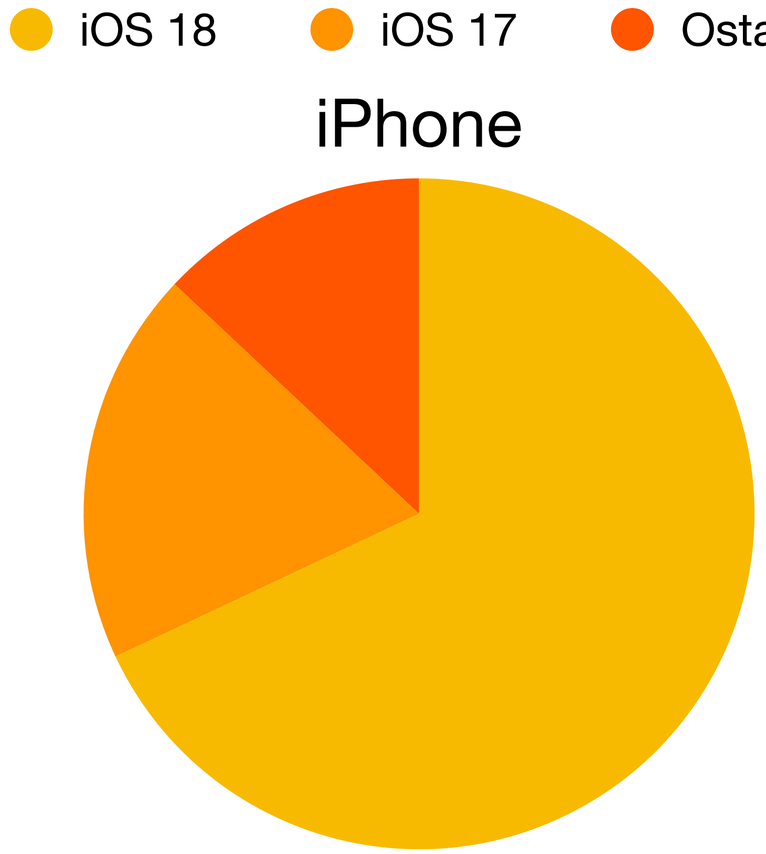
Životní cyklus iOS

- Nové verze OS vydávány ročně (beta v červnu na WWDC, public v září spolu s novým HW).
- Během roku minor/patch verze.
- Podpora až 5 let.
- Do měsíce po vydání mívá verzi 50 % zařízení.
- Od iOS 14/15 Apple nenutí k upgradu.

Zastoupení verzí iOS



Max. 4 roky staré zařízení



Zastoupení u všech zařízení

Zdroj: https://developer.apple.com/support/app-store/

Distribuce aplikací

- App Store (Developer Program za $99).
  - Provize Applu 30% (15%).
  - Každá aplikace musí projít review.
  - Je možnost si vyžádat i zrychlené review
- TestFlight, interní a externí testeři, public link.
- Interně (Enterprise Program za $299).

Způsoby monetizace

- Zdarma nebo placené.
- Jednorázové nákupy v aplikaci
- Předplatné

Přístupnost

- Guided Access
- VoiceOver
- Dynamic Type
- Grafika (tmavý režim, vysoký kontrast, inverzní barvy, snížená průhlednost, snížená barevnost, omezené animace)

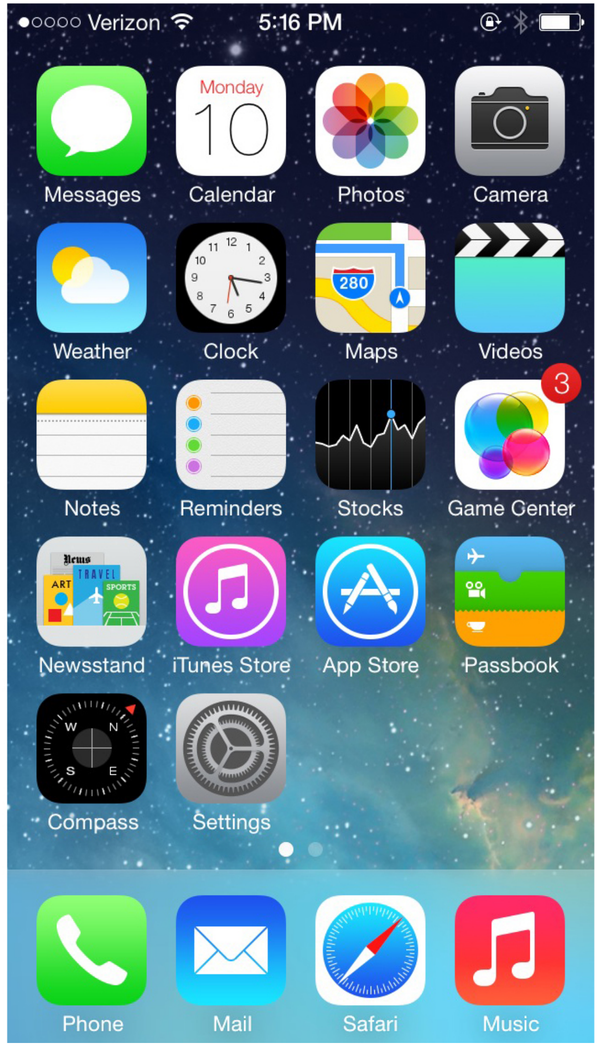
Bezpečnost

- Sandboxing
- Potvrzování přístupů jednotlivě
- Keychain
- App Transport Security (2015)
- Passcode, TouchID (2013), FaceID (2017)
- Šifrování obsahu (v HW)
- Apple ID účet s 2FA

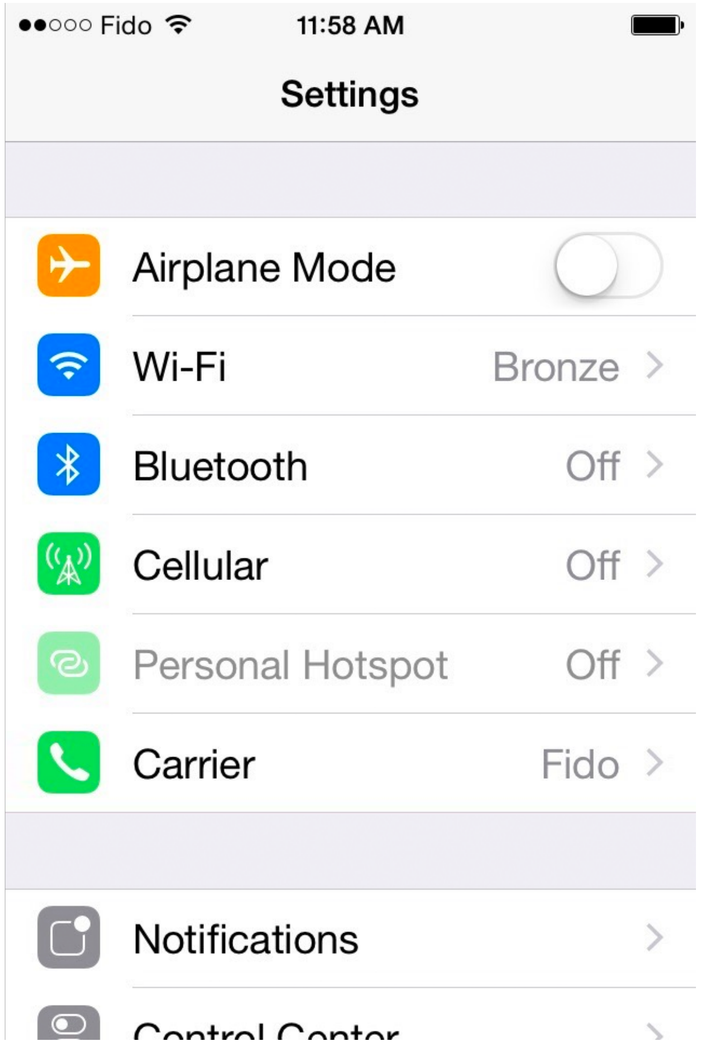
Uživatelské rozhraní

- Human Interface Guidelines
- Preference systémových komponent
- Primárně dotykové, lze mít periferie
- Widgety
- Notifikace
- Skeuomorphism → Flat design (od iOS 7 z 2013)

Skeuomorphism → Flat design



Skeuomorphism → Flat design

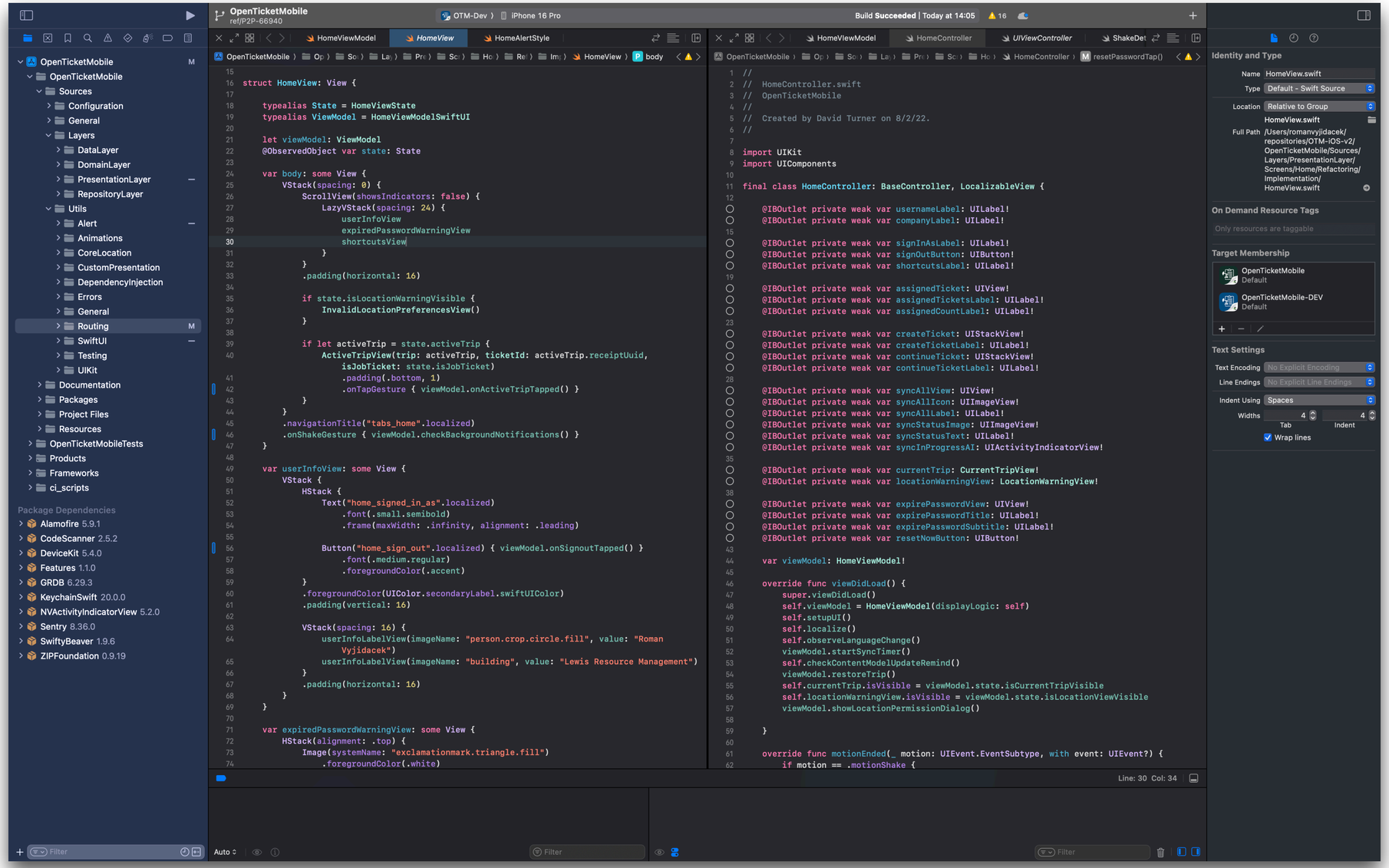


Vývojové prostředí

Xcode

- Vývojové prostředí (IDE) pro vývoj aplikací na platformách firmy Apple
- Hlavní funkce zahrnují:
  - Návrh uživatelského rozhraní (UI)
  - Psaní kódu v jazyce Swift nebo Objective-C
  - Ladění aplikací
  - Testování na simulátorech nebo fyzických zařízeních

Xcode

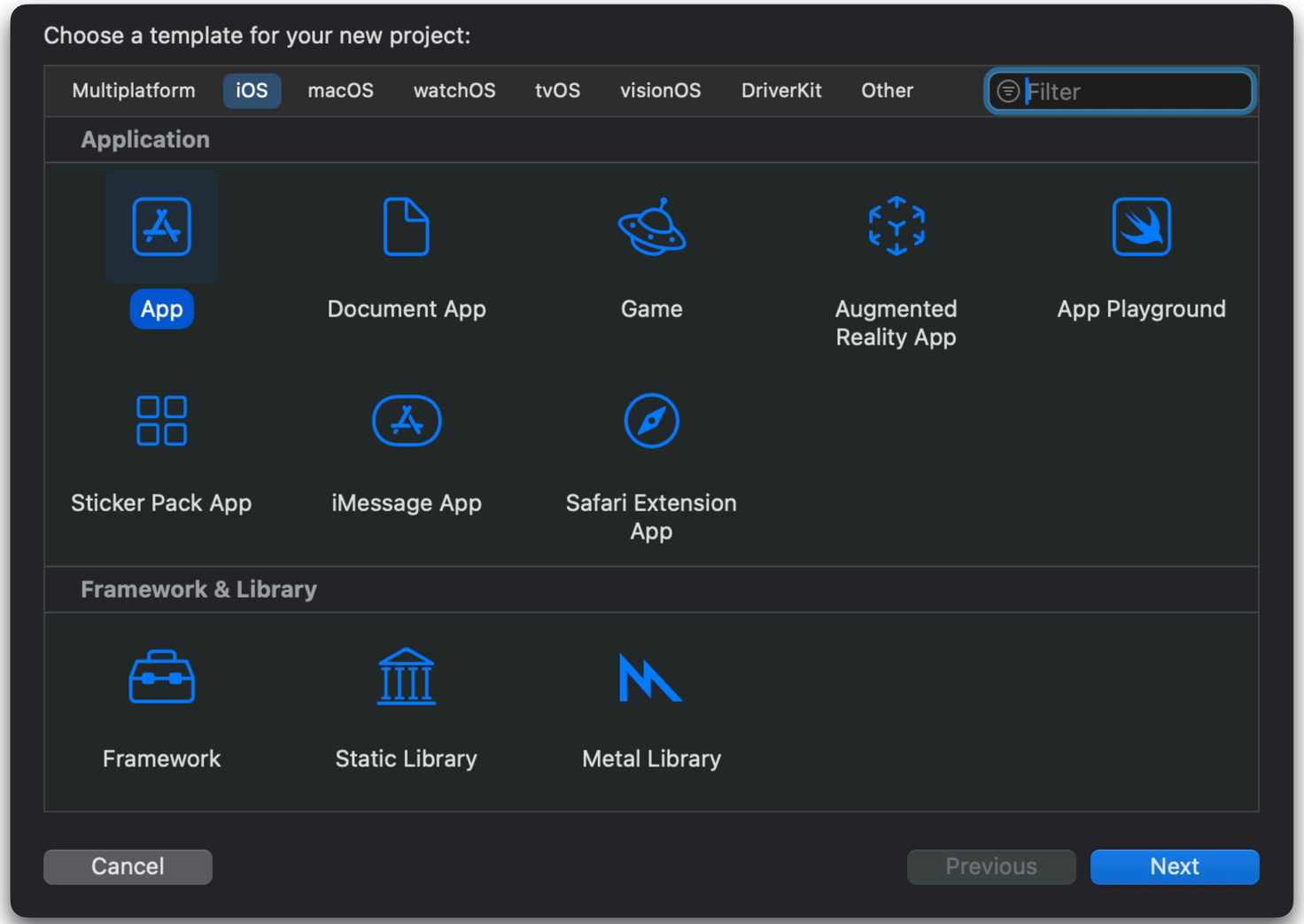

OpenTicketMobile
ref/P2P-66940
OTM-Dev
iPhone 16 Pro
Build Succeeded | Today at 14:05
HomeViewModel
HomeView
HomeAlertStyle
HomeController
UIViewController
Identity and Type
Name HomeView.swift
Type Default - Swift Source
Location Relative to Group
Full Path /Users/romanvyjidacek/repositories/OTM-iOS-v2/OpenTicketMobile/Sources/Layers/PresentationLayer/Screens/Home/Refactoring/Implementation/HomeView.swift
On Demand Resource Tags
Only resources are taggable
Target Membership
OpenTicketMobile
Default
OpenTicketMobile-DEV
Default
Text Settings
Text Encoding No Explicit Encoding
Line Endings No Explicit Line Endings
Indent Using Spaces
Widths 4 4
Tab Indent
Wrap lines

Vytvoření projektu

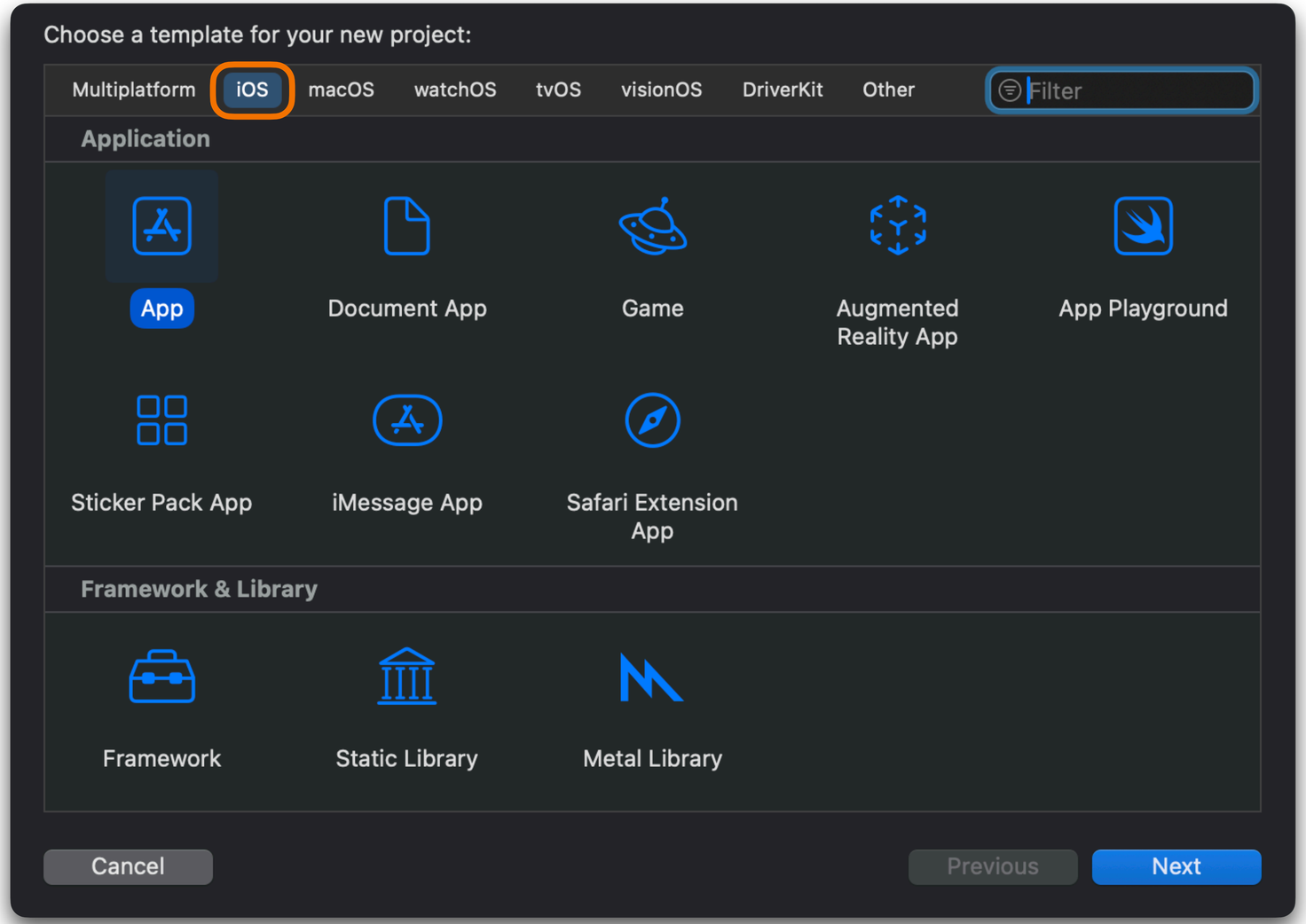
Vytvořit nový projekt

Vytvořit nový projekt

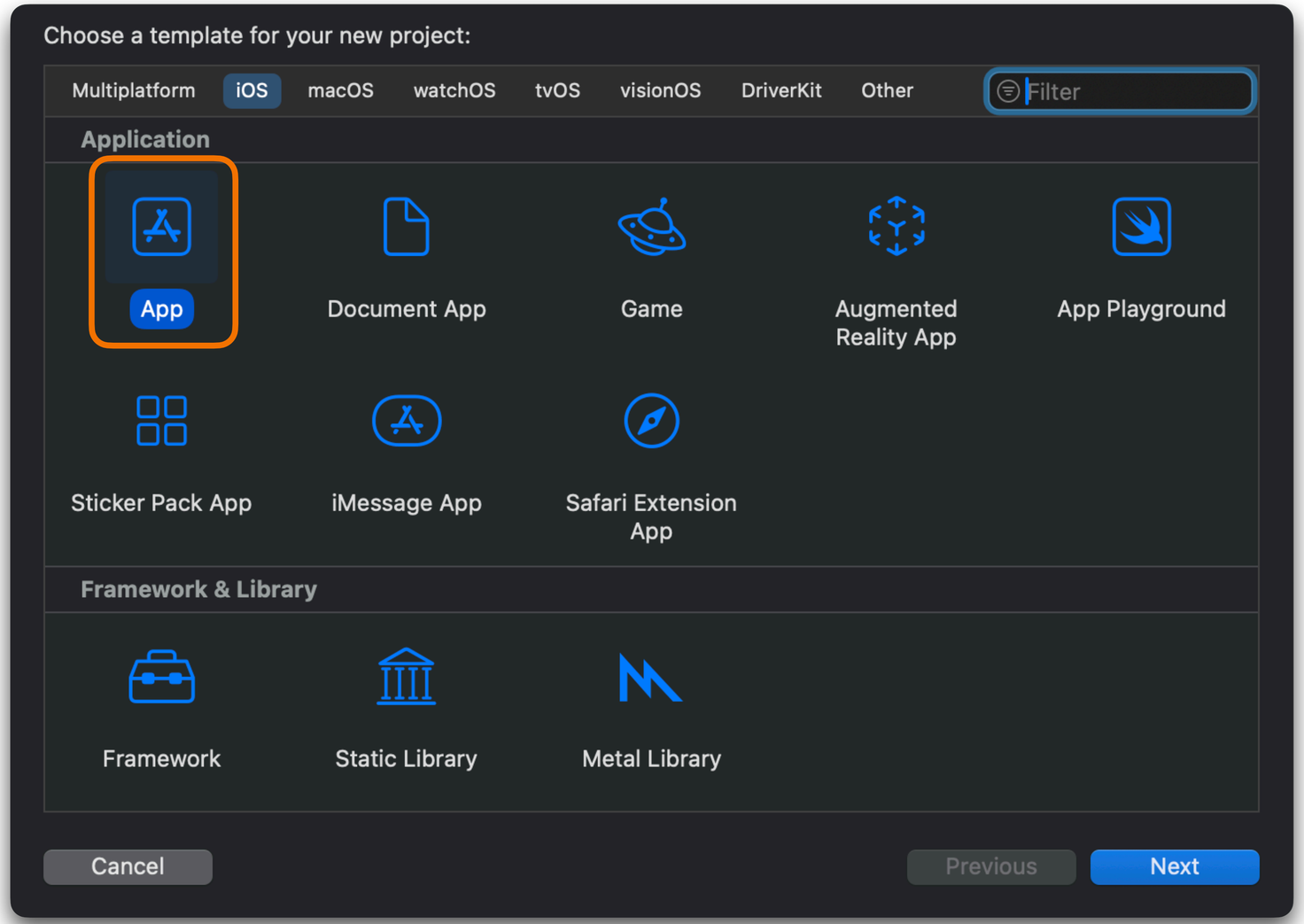
Vybereme typ aplikace



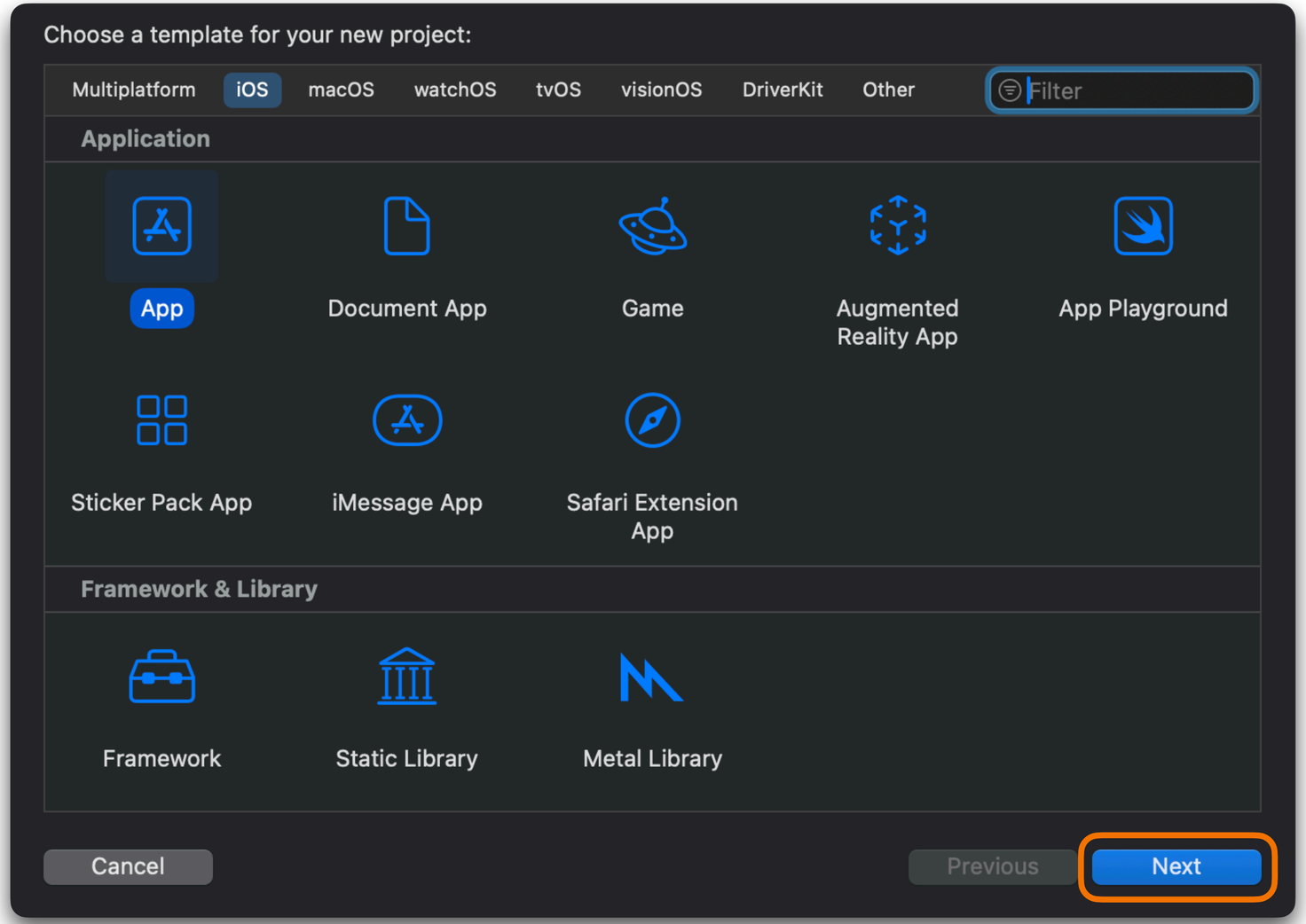
Vybereme typ aplikace



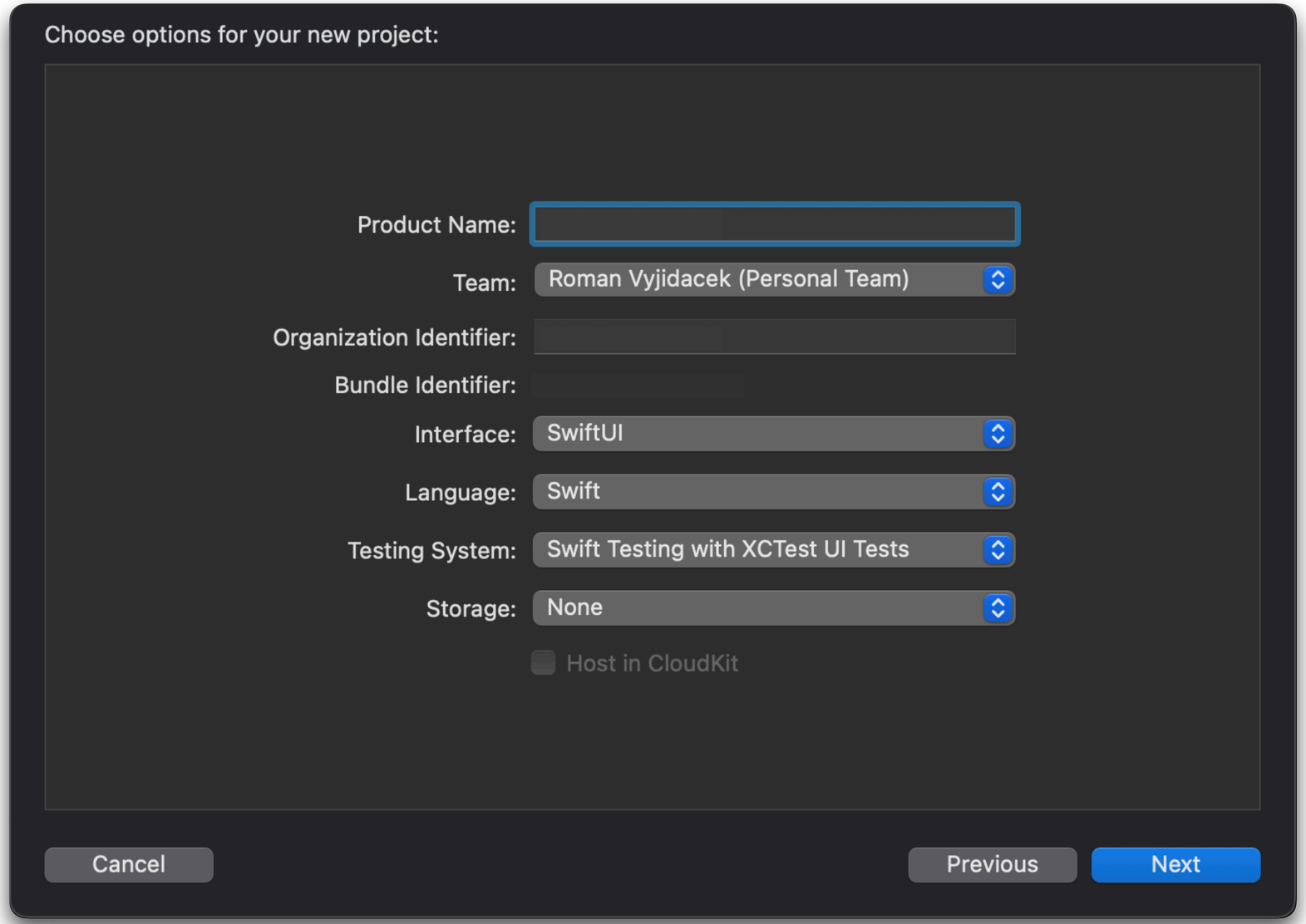
Vybereme typ aplikace



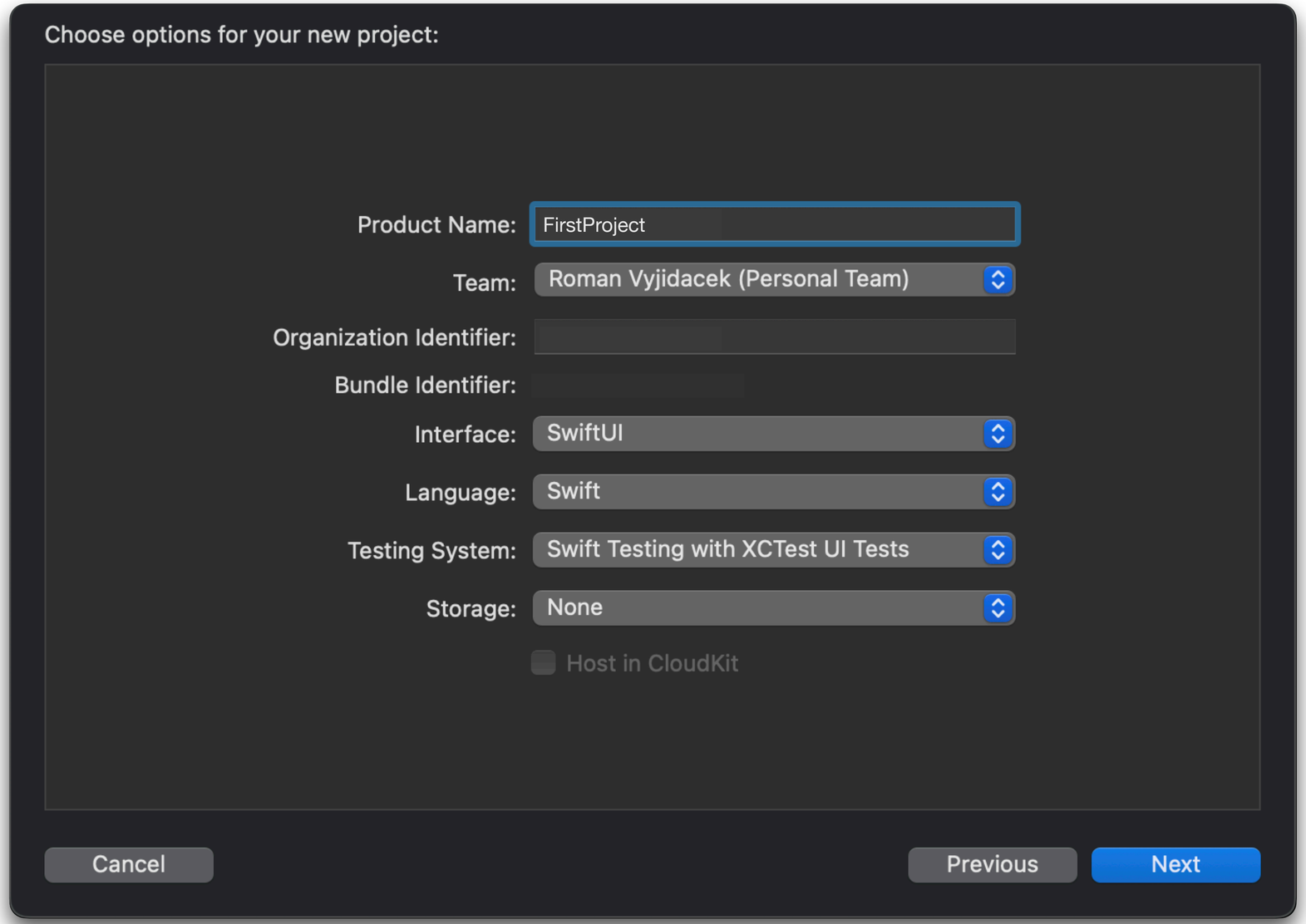
Vybereme typ aplikace



Nastavení projektu



Nastavení projektu



Product Name v Xcode by měl být jedinečný, krátký, bez mezer a speciálních znaků, a začínat velkým písmenem.

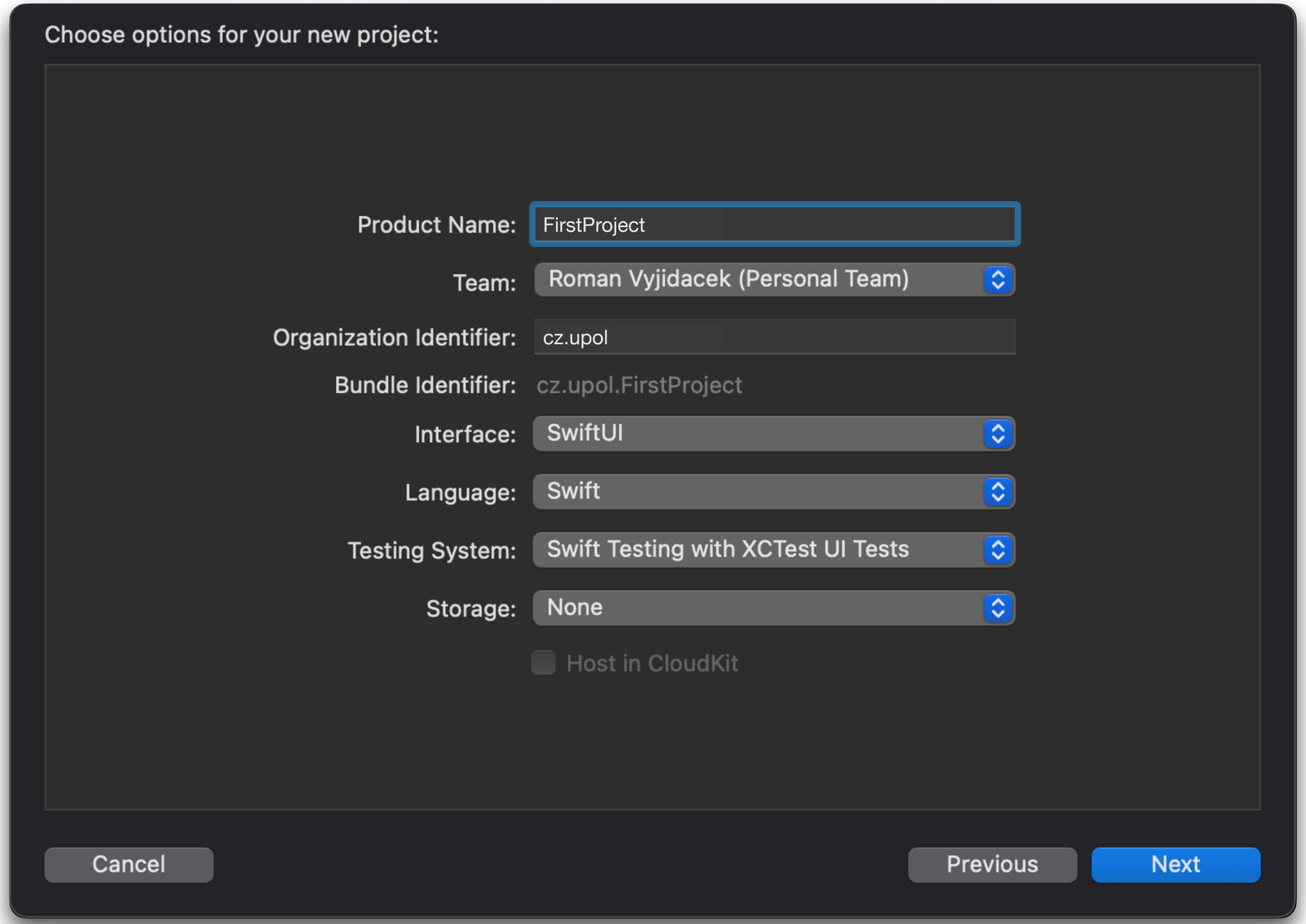
Příklad správných názvů:

✅ WeatherTracker | ✅ ToDoApp

Příklad špatných názvů:

❌ 123MyApp | ❌ My App

Nastavení projektu



Product Name v Xcode by měl být jedinečný, krátký, bez mezer a speciálních znaků, a začínat velkým písmenem.

Příklad správných názvů:

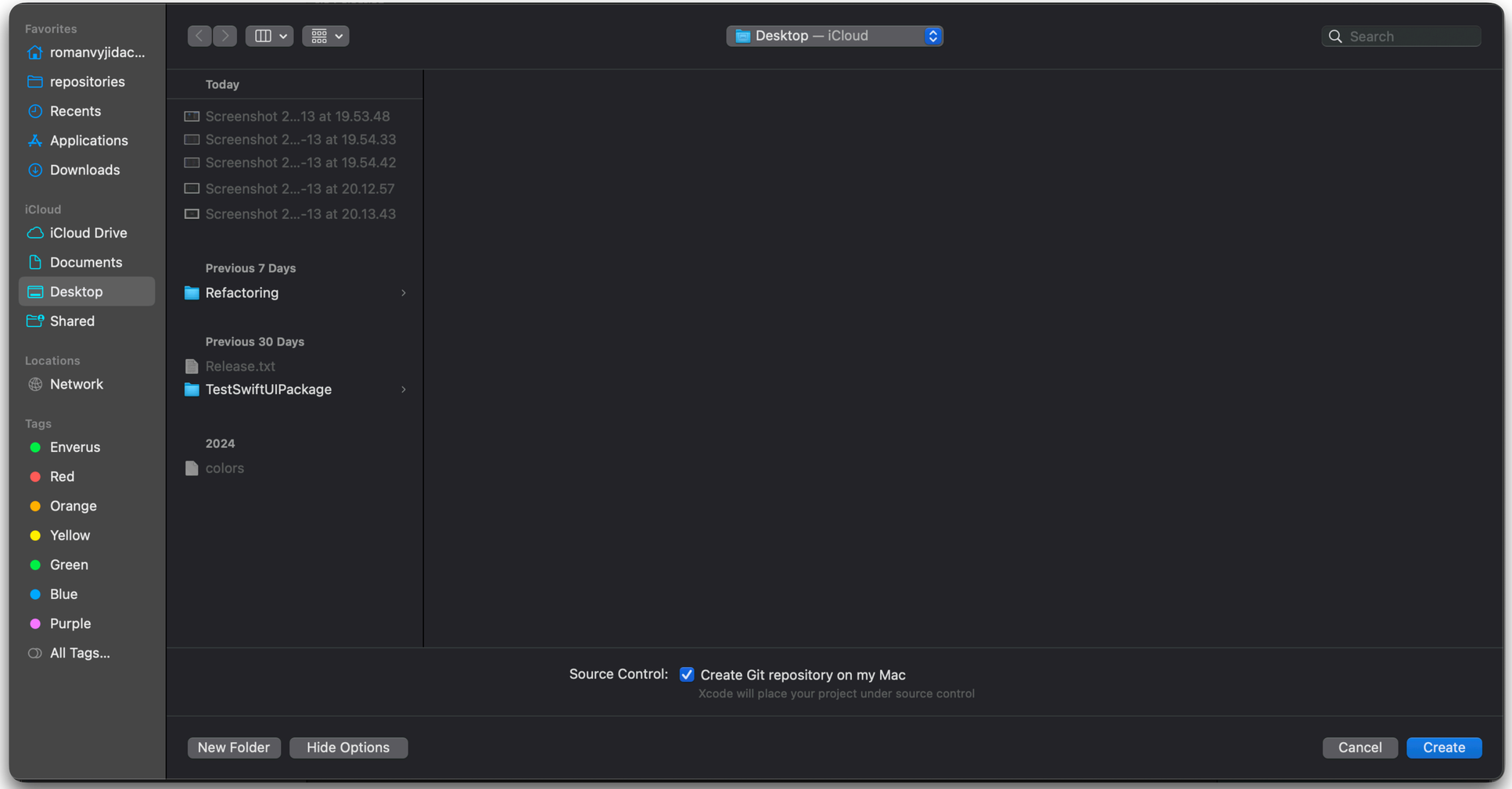
✅ WeatherTracker | ✅ ToDoApp

Příklad špatných názvů:

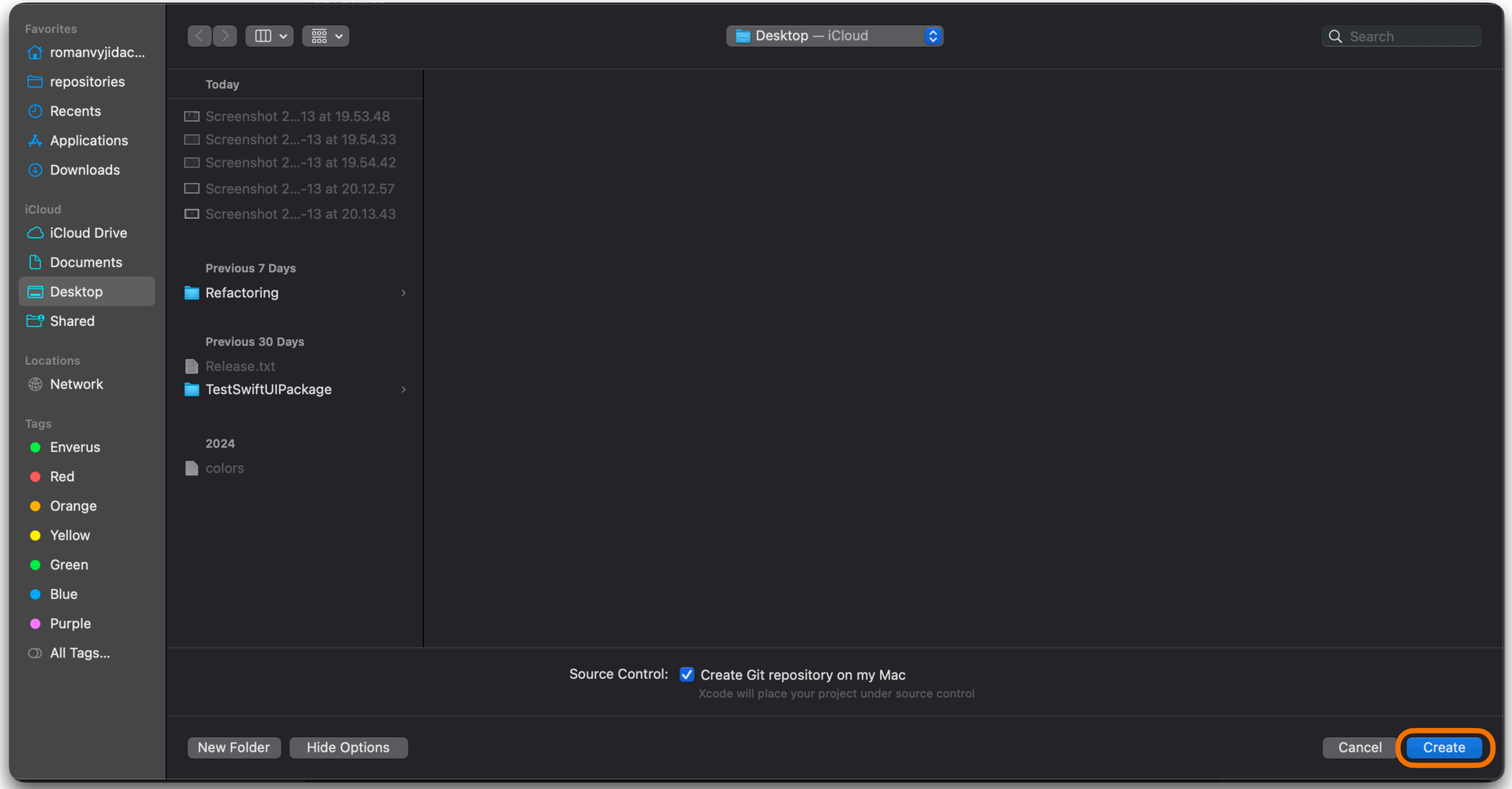
❌ 123MyApp | ❌ My App

Organization Identifier je součástí Bundle Identifier, který jednoznačně identifikuje aplikaci v App Store a na zařízeních. Používá se v reverse domain notation.

Vybereme adresář



Vybereme adresář



Základy jazyka Swift

Swift

- Kompletně nový jazyk
- Uveden 6/2014, vývoj od 2010
- Každý rok nová major verze
- Zpočátku radikální vývoj rozbíjel kompatibilitu od verze 2 open source pod Apache licencí
- Kompletní dokumentace: https://docs.swift.org/swift-book/documentation/the-swift-programming-language/

Balíčkovací systémy

- Swift Package Manager
  - Nativní řešení, získává na oblibě, decentralizovaný
  - Správa přes UI
- Cocoapods
  - Nejrozšířenější správce závislostí
  - Centralizované úložiště, používá **Podfile** (soubor s deklarací závislostí)
- Carthage
  - Lehká a flexibilní alternativa
  - Decentralizovaný přístup, generuje frameworky namísto integrace kódu.

Hello World!

- Začneme klasicky s programem “Hello World!”
- Spuštení programu:
  - Terminál: `swift main.swift`
  - Swift Playground: https://developer.apple.com/swift-playground/
  - Online: https://www.programiz.com/swift/online-compiler/

```
// Vytiskne řetězec + \n
print("Hello, World!")

// Vytiskne pouze řetězec
print("Hello World!",
      terminator: "")
```

Proměnné a konstanty

- `var` definuje proměnnou, jejíž hodnota se může změnit
- `Let` definuje konstantu, která se nemůže změnit
- Pro názvy používáme camelCase
- Není nutné používat ;
- Swift používá typovou inferenci

```
var name = “Alice"

let age = 25
```

Datové typy

- Swift má silnou typovou kontrolu.
- Nemůžeme tedy například provádět aritmetické operace mezi `Int` a `Double`
- Hlavní typy: `Int`, `Double`, `Bool`, `String`
- Při definici proměnné můžeme (a je to běžné) i datový typ

```
var number: Int = 10

var price: Double = 99.99

var isSwiftFun: Bool = true

var message: String = “Hi"
```

Podmínky

- Podmínky používají relační operátory: ==, !=, <, >, <=, >=
- Nejsou potřeba závorky (…) kolem podmínky
- Složitější podmínky je možné vytvářet pomocí logických operátorů && (AND) a || (OR)

```
let age = 18
let isCarAvaliable = false

if age >= 18 {
    print("Můžeš řídit.")
} else {
    print("Nemůžeš řídit.")
}

// Složitější podmínka
if age >= 18 && isCarAvaliable {
    print("Můžeš řídit.")
} else {
    print("Nemůžeš řídit.")
}
```

Switch

- Místo zanořených `if-else` bloků, kde pouze vybíráme jednu hodnotu je lepší využít konstrukce `switch`
- Je možné “přepínat” všechny základní datové typy + výčtové datové typy
- `Switch` musí pokrýt všechny možné případy nebo mít default větev
- Zde je prezentován pouze základní výraz → více v dokumentaci

```
let grade = "B"

switch grade {
case "A":
    print("Výborně!")
case "B":
    print("Dobrá práce!")
default:
    print("Zkus to lépe.")
}
```

Cykly (for, while)

- `For-in` iteruje přes sekvence (`Array`, `Range`)
- `While` se opakuje, dokud platí podmínka
- `Repeat-while` nejprve provede blok kódu, poté kontroluje podmínku
- Použijeme `for-in`, pokud víš, kolikrát se má smyčka opakovat. Použijeme `while`, pokud to nevíme předem.

```
// Čísla 1,2,3,4,5
for number in 1...5 {
    print("Číslo: \(number)")
}

// Čísla 1,2,3,4
for number in 1..<5 {
    print("Číslo: \(number)")
}

var count = 3

while count > 0 {
    print(count)
    count -= 1
}
```

Funkce

- `func` definuje funkci
- Pro názvy používáme camelCase
- Parametry mají pevně daný typ
- Funkce mohou mít výchozí hodnoty parametrů
- Pokud chceme předat hodnotový typ odkazem je nutné definovat parametr jako `inout`

```
func greet(name: String) -> String {
    return "Ahoj, \(name)!"
}

greet(name: "Petr")

func sayHello(name: String = "světe") {
    print("Ahoj, \(name)!")
}

sayHello()
sayHello(name: “Jana")

func changeValue(number: inout Int) {
    number = Int.random(in: 0...100)
}

var number = 2
changeValue(number: &number)
```

Kolekce

- Pole je datový typ předávaný hodnotou
- Ve skutečnosti však Swift předává kolekce odkazem a při zápisu vytvoří kopii
- Přes kolekce lze snadno iterovat pomocí `for-in` cyklu
- Slovník ukládá data ve formátu klíč-hodnota

```
var fruits = ["Jablko", “Banán",
              "Hruška"]
// Přístup k prvnímu prvku
fruits[0]
// Přidání prvku
fruits.append("Broskev")
// Odstranění druhého prvku
fruits.remove(at: 1)
// Počet prvků
fruits.count

var person = ["Sheldon": "Dr.",
              "Wolowitz": "Mr."]
// Získáme hodnotu
person["Sheldon"]
// Odstraníme hodnotu
person["Sheldon"] = nil
```

Struktury

- `Struct` je hodnotový typ – při přiřazení nebo předání funkci se kopíruje
- Používá se pro jednoduchá data, jako jsou modely, souřadnice, nastavení atd
- `Mutating` umožňuje měnit vlastnosti struktury uvnitř metody.
- Neumožňuje dědičnost – každý struct je samostatný
- Struktury jsou thread-safe

```
struct Point {
    var x: Int
    var y: Int

    mutating func moveBy(x: Int, y: Int) {
        self.x += x
        self.y += y
    }
}

var p = Point(x: 3, y: 4)
p.moveBy(x: 2, y: -1)
print(p) // Point(x: 5, y: 3)
```

Třídy

- `Class` je referenční typ – při přiřazení se nepřekopíruje, ale předává se reference.
- Umožňuje dědičnost (inheritance) – může mít nadřazenou třídu.
- Změny v objektu se projeví všude, kde je reference na instanci.

```
class Animal {
    var name: String

    init(name: String) {
        self.name = name
    }

    func makeSound() {
        print("Nějaký zvuk")
    }
}

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

let dog = Dog(name: "Rex")
dog.makeSound() // Haf haf!
```

Shrnutí struct vs. class

Vlastnost	struct (Struktura)	class (Třída)
Typ	Hodnotový (Value Type)	Referenční (Reference Type)
Kopírování	Vytvoří novou kopii	Předává referenci
Dědičnost	❌ Nepodporuje	✅ Podporuje
Změna vlastností	Je potřeba `mutating`	Není potřeba `mutating`
Thread-safe	Ano – každá instance je unikátní	Možné problémy – reference mohou být sdílené

Computed properties

- Používáme pro hodnoty, které můžeme vypočítat z ostatních vlastností
- `var area` není ukládána do paměti, ale vypočítává se při každém přístupu
- Může mít také set blok pro aktualizaci hodnoty
- Computed properties používáme místo funkcí, pokud se hodnota logicky vztahuje k objektu.

```
class Rectangle {
    var width: Double
    var height: Double

    var area: Double {
        return width * height
    }

    var perimeter: Double {
        get { return 2 * (width + height) }
        set { width = newValue / 4
              height = newValue / 4 }
    }

    init(width: Double, height: Double) {
        self.width = width
        self.height = height
    }
}

let rect = Rectangle(width: 5, height: 10)
print(rect.area) // 50.0
```

Optionals

- Umožňují proměnným nemít žádnou hodnotu (`nil`)
- Swift neumožňuje nil v běžných proměnných – musí být Optional
- Pokud chceme hodnotu “rozbalit” tak použijeme !
- Nikdy nepoužívejte ! bez kontroly, zda hodnota existuje!
- Pomocí operátoru ?? Můžeme definovat výchozí hodnotu pokud je hodnota `nil`

```swift
var name: String? = nil
name = "Alice"
print(name) // Optional("Alice")

// Alice (ale může způsobit
crash, pokud je nil!)
print(name!)

var username: String? = nil
let user = user ?? "Vader"
print(user) // "Vader"
```

Bezpečné rozbalení Optional

- `If let` bezpečně rozbalí Optional a použije jeho hodnotu
- Pokud je hodnota `nil`, provede se else (je-li přítomna) blok
- `Guard let` Slouží ke kontrole podmínek na začátku funkce
- Pokud podmínka není splněna, kód okamžitě opustí funkci (return)

```swift
var name: String? = "Alice"

if let unwrappedName = name {
    print("Ahoj, \(unwrappedName)")
} else {
    print("Žádné jméno")
}

func greet(name: String?) {
    guard let unwrappedName = name else {
        print("Jméno chybí!")
        return
    }
    print("Ahoj, \(unwrappedName)")
}
```

Výčtové typy

- Výčtový typ (enum) umožňuje definovat sadu souvisejících hodno
- Každá hodnota (case) je unikátní a typově bezpečná
- Enum může obsahovat asociované hodnoty a výchozí hodnoty (raw values)
- V enum je možné definovat metody
- Samostudium: rawValues, switch pro výčtové typy

```
enum Direction {
    case north
    case south
    case east
    case west
}

let move = Direction.north
var current: Direction = .east

enum Barcode {
    case upc(Int, Int, Int, Int)
    case qrCode(String)
}

let upc = Barcode.upc(8,85909,51226,3)
let qr = Barcode.qrCode("ABC123XYZ")
```

Výjimky a jejich zpracování

- Typ výjimky definujeme jako `enum` implementující protokol `Error`
- Funkci která může vyvolat výjimku označíme jako `throws`
- Pomocí `throw` vyvoláme výjimku
- V `do-catch` zachytíme chybu a umožníme její zpracování
- Můžeme však výjimku ignorovat pomocí `try?`

```
enum PasswordError: Error {
    case tooShort
}

func checkPassword(_ password: String) throws {
    if password.count < 6 {
        throw PasswordError.tooShort
    }
}

do {
    try checkPassword("123")
} catch {
    print("Chyba: heslo je příliš krátké.")
}

// Vrátí `nil` při chybě
let result = try? checkPassword("123")
```

Protokol

- Definuje rozhraní, které musí třída/ struktura splnit
- Nevlastní žádnou implementaci – pouze požadavky
- Pomocí extension (viz dále) ale můžeme vytvořit defaultní implementaci

```
protocol Drivable {
    var speed: Int { get set }
    func drive()
}

class Car: Drivable {
    var speed = 100

    func drive() {
        print("Auto jede rychlostí
              \(speed) km/h.")
    }
}
```

Extensions

- Extensions přidávají nové metody k existujícím typům bez nutnosti dědičnosti

```
extension Int {

    func squared() -> Int {
        return self * self
    }
}

let number: Int = 4
print(number.squared())
```

Samostudium

Generika
Closures
Funkce vyšších řádů:
  - map, reduce, filter
Makra (základy)

Úkoly

- Termín odevzdání je na následujícím semináři (2. dubna)
- Konzultace (MS Teams) řešení možná po předchozí domluvě emailem
- Zkuste se vyhnout použití AI (ChatGPT, …)

Úkol 1: Správa objednávek

- Vytvořte PaymentMethod, který bude obsahovat možnosti cash, card, applePay.
- Vytvořte model pro Order, který bude obsahovat:
  - id: Int
  - amount: Double (částka objednávky)
  - paymentMethod: PaymentMethod
- Vytvořte seznam objednávek (Array<Order>).
- Bez použití cyklů získejte: pouze objednávky placené kartou, seznam částek všech objednávek a celkový příjem ze všech objednávek.

Úkol 2: Generická datová struktura

- Vytvořte generickou třídu Storage.
- Přidejte do ní metody:
  - add(_ item: T): Přidá prvek do úložiště.
  - filterItems(_ condition: (T) -> Bool) -> [T]: Vrátí prvky splňující podmínku.
- Umožněte použití pouze pro typy, které implementují Equatable.
- Vytvořte instanci Storage<String> a přidejte do ní několik slov.
- Použijte filterItems, aby byla získána slova delší než 5 znaků.

Úkol 3: Implementace univerzální fronty

- Vytvořte protokol QueueProtocol, který bude definovat metody:
  - enqueue(_ item: T): Přidá prvek do fronty.
  - dequeue() -> T?: Odebere prvek z fronty (FIFO).
  - peek() -> T?: Vrátí první prvek bez odebrání.
- Implementujte generickou třídu Queue<T>, která bude tento protokol dodržovat.
- Omezte Queue<T> pouze na typy, které splňují Equatable.
- Přidejte do třídy možnost získat počet prvků ve frontě.
- Vytvořte dvě instance Queue s různými datovými typy (Int a String).
- Otestujte přidání (enqueue), odebrání (dequeue) a náhled (peek) prvků.

Úkol 4: Systém správy úkolů

- Vytvořte Task, která bude obsahovat:
  - id: String (hint: UUID), status: TaskStatus, deadline: Date?
  - TaskStatus nabývá hodnot [inProgress, completed, pending]
- Implementujte TaskManager, která bude spravovat seznam úkolů a poskytovat metody:
- addTask(_ task: Task)
- completeTask(id: String)
- completeNextTask() (Vykoná Task s nejvyšší prioritou)
- tasksInProgress() -> [Task]
- highestPriorityTask() -> Task?
- Přidejte několik úkolů do TaskManager a simulujte změnu stavu (completeTask, completeNextTask)