Lightsaber App Teaching Outline

Course Overview

This hands-on workshop teaches Swift networking fundamentals through building a Star Wars-themed lightsaber management app. Students will learn core concepts like HTTP requests, JSON handling, loading states, and error management in a fun, engaging context.

Target Audience: Swift beginners with basic iOS development knowledge
Duration: 2 hours (intensive hands-on session)
Prerequisites: Basic Swift syntax, SwiftUI fundamentals

Learning Objectives

By the end of this workshop, students will understand:

- How functions can be passed around as parameters (higher-order functions)
- Making HTTP requests (GET, POST, PATCH, DELETE)
- JSON encoding and decoding in Swift
- Managing loading states in iOS apps
- Implementing proper error handling
- Introduction to authentication concepts (tokens)

Module 1: Functions as First-Class Citizens (15 minutes)

Concept: "Functions Can Be Passed Around"

In Swift, functions are "first-class citizens" - they can be stored in variables, passed as parameters, and returned from other functions.

Learning Activities:

1.1 Simple Function Parameter Example

```
// Start with a simple example students can relate to
func greetJedi(name: String, formatter: (String) -> String) -> String {
    return formatter("Hello, \(name)")
}

let upperCaseFormatter: (String) -> String = { text in
    return text.uppercased()
}

let result = greetJedi(name: "Luke", formatter: upperCaseFormatter)
// Result: "HELLO, LUKE"
```

1.2 More Complex Example

Show how functions can be stored and passed around:

```
// Different ways to format lightsaber data
let formatters: [(Lightsaber) -> String] = [
    { lightsaber in "\(lightsaber.name) - \(lightsaber.color)" },
    { lightsaber in "\(lightsaber.creator)'s \(lightsaber.color) blade" },
    { lightsaber in "A \(lightsaber.crystalType) crystal powers this \(lightsaber.color
]

func displayLightsaber(_ lightsaber: Lightsaber, using formatter: (Lightsaber) -> Strin
    return formatter(lightsaber)
}

// Usage
let luke = Lightsaber(name: "Luke's Saber", color: "blue", creator: "Luke", crystalType
let result = displayLightsaber(luke, using: formatters[0])
// Result: "Luke's Saber - blue"
```

1.3 Hands-On Exercise

- Create a simple function that takes another function as a parameter
- Apply it to format lightsaber names (uppercase, lowercase, title case)

Key Takeaway: Functions are just like any other data type in Swift - you can pass them around!

Module 2: HTTP Methods & JSON Fundamentals (30 minutes)

2.1 Understanding HTTP Methods (10 minutes)

Learn the essential HTTP methods through lightsaber CRUD operations.

Method	Purpose	Lightsaber Example
GET	Retrieve data	Get all lightsabers
POST	Create new data	Add new lightsaber
PUT	Replace entire data	Replace whole lightsaber
PATCH	Update partial data	Modify lightsaber details
DELETE	Remove data	Delete a lightsaber

2.2 JSON Encoding & Decoding (20 minutes)

Understanding JSON Structure (5 minutes)

Show the API response format:

```
{
  "success": true,
  "data": [
    {
      "id": "123",
      "name": "Luke's Lightsaber",
      "color": "blue",
      "creator": "Luke Skywalker",
      "crystalType": "Kyber",
      "hiltMaterial": "Durasteel",
      "createdAt": "2024-07-25T10:30:00Z",
      "isActive": true
    }
  ],
  "message": null
}
```


Swift Codable Protocol (5 minutes)

Examine the `Lightsaber` model:

```
struct Lightsaber: Identifiable, Codable {
    let id: String
    let name: String
    let color: String
    // ... other properties
}
```

Key points:

- `Codable = Encodable + Decodable`
- Automatic JSON mapping for matching property names
- Custom coding keys for different JSON field names

Hands-On Exercise (10 minutes)

Students practice:

1. Decode a sample JSON string to `Lightsaber` object
2. Encode a `Lightsaber` object to JSON data
3. Handle the `APIResponse` wrapper structure

```swift
// Example exercise
let jsonString = """
{
    "id": "456",
    "name": "Yoda's Lightsaber",
    "color": "green"
}
"""

let jsonData = jsonString.data(using: .utf8)!
let lightsaber = try JSONDecoder().decode(Lightsaber.self, from: jsonData)
```

Key Takeaway: Codable makes JSON handling almost magical - Swift does the heavy lifting!

Module 3: HTTP Request Implementation (50 minutes)

3.1 Implementing GET Request (20 minutes)

Theory (5 minutes)

- URL structure: `http://localhost:3000/api/lightsabers`
- Headers and response handling
- URLSession basics

Practice (15 minutes)

Students implement `fetchLightsabers()` in `LightsaberService.swift`:

```
func fetchLightsabers() async {
    isFetchingLightsabers = true
    errorMessage = nil

    guard let url = URL(string: "\(baseURL)/lightsabers") else {
        errorMessage = "Invalid URL"
        isFetchingLightsabers = false
        return
    }

    do {
        let (data, _) = try await URLSession.shared.data(from: url)
        let response = try JSONDecoder().decode(APIResponse<[Lightsaber]>.self, from: d
        lightsabers = response.data
    } catch {
        errorMessage = error.localizedDescription
    }

    isFetchingLightsabers = false
}
```

3.2 Implementing POST Request (20 minutes)

Theory (5 minutes)

- Request body and headers
- Content-Type: application/json
- Status codes (201 Created)

Practice (15 minutes)

Students implement `createLightsaber()`:

```
func createLightsaber(_ lightsaber: Lightsaber) async -> Bool {
    isCreatingLightsaber = true
    errorMessage = nil

    guard let url = URL(string: "\(baseURL)/lightsabers") else {
        errorMessage = "Invalid URL"
        isCreatingLightsaber = false
        return false
    }

    var request = URLRequest(url: url)
    request.httpMethod = "POST"
    request.setValue("application/json", forHTTPHeaderField: "Content-Type")

    do {
        let jsonData = try JSONEncoder().encode(lightsaber)
        request.httpBody = jsonData

        let (data, response) = try await URLSession.shared.data(for: request)

        if let httpResponse = response as? HTTPURLResponse,
           httpResponse.statusCode == 201 {
            let newLightsaber = try JSONDecoder().decode(Lightsaber.self, from: data)
            lightsabers.append(newLightsaber)
            isCreatingLightsaber = false
            return true
        } else {
            errorMessage = "Failed to create lightsaber"
            isCreatingLightsaber = false
            return false
        }
    } catch {
        errorMessage = error.localizedDescription
        isCreatingLightsaber = false
        return false
    }
}
```

3.3 Implementing PATCH & DELETE (10 minutes)

Students complete one of the remaining methods following similar patterns:

- `updateLightsaber()` - PATCH request OR
- `deleteLightsaber()` - DELETE request

Key Takeaway: HTTP methods are just different ways to communicate what action you want to perform on the server.

Module 4: Loading States & Error Handling (15 minutes)

Concept: Providing User Feedback During Network Operations

Users need to know when the app is working. Loading states prevent confusion and improve user experience.

4.1 Loading States (5 minutes)

Examine the service's loading properties:

```
@Published var isFetchingLightsabers = false
@Published var isCreatingLightsaber = false
@Published var isUpdatingLightsaber = false
@Published var isDeletingLightsaber = false
```

Why separate states?

- Different operations can happen simultaneously
- Specific UI feedback per action
- Better user experience

4.2 Error Handling (10 minutes)

Quick overview of error handling strategy:

```
@Published var errorMessage: String?

// In API methods:
do {
    // API call
} catch {
    errorMessage = error.localizedDescription
}
```

Show error display in UI:

```
.alert("Error", isPresented: .constant(service.errorMessage != nil)) {
    Button("OK") {
        service.errorMessage = nil
    }
} message: {
    Text(service.errorMessage ?? "")
}
```

Key Points:

- Always handle errors in network operations
- Show loading states during operations
- Provide user-friendly error messages

Module 5: Authentication Teaser (10 minutes)

Concept: Introduction to Token-Based Authentication

While not implemented in this app, students learn the concepts for future projects.

5.1 What is Authentication? (3 minutes)

- Verifying user identity
- Protecting sensitive data
- Controlling access to resources

5.2 Token-Based Authentication Flow (5 minutes)

1. **Login**: User provides credentials (username/password)
2. **Token Generation**: Server returns authentication token
3. **Token Storage**: App stores token securely
4. **Authenticated Requests**: Include token in request headers

Example header:

```
request.setValue("Bearer \(authToken)", forHTTPHeaderField: "Authorization")
```

5.3 Security Considerations (2 minutes)

- Never store passwords
- Use secure storage for tokens (Keychain)
- Handle token expiration gracefully

Key Takeaway: Authentication is crucial for real apps - tokens are like temporary keys to access user data.

Assessment & Practice Activities

Quick Checks (Throughout Workshop)

- "Explain what happens when we call `fetchLightsabers()` "
- "Why do we need separate loading states?"
- "What would happen if we forgot error handling?"

Final Project Challenge

Students enhance the app by:

1. Adding search functionality (GET with query parameters)
2. Implementing data caching
3. Adding retry logic for failed requests
4. Creating better error messages

Code Review Session

- Students present their implementations
- Discuss different approaches and trade-offs
- Address common mistakes and improvements

Resources & Next Steps

Additional Learning

- Swift Async/Await deep dive
- Combine framework for reactive programming
- Network layer architecture patterns
- Unit testing network code

Tools & Documentation

- Postman for API testing
- Charles/Proxyman for network debugging
- Apple's URLSession documentation
- Swift.org Codable guide

Teaching Tips

For Instructors

Make it Interactive:

- Use live coding demonstrations
- Encourage questions throughout
- Pair programming for exercises
- Code review sessions

Common Student Mistakes:

- Forgetting to handle errors

- Not updating loading states properly
- Incorrect JSON structure assumptions
- Missing async/await keywords

Engagement Strategies:

- Star Wars references and humor
- Real-world examples beyond the app
- "What if..." scenarios for error cases
- Connect concepts to apps they use daily

Pacing Adjustments:

- Spend more time on concepts students struggle with
- Skip advanced topics if running short on time
- Have extension activities for fast learners
- Provide code templates for struggling students

This outline balances theory with hands-on practice, ensuring students not only learn the concepts but can apply them confidently in their own projects.