What I've Learn[illegible]d in the Fourth Module (UI + Presentation)

- App UI, UX, Localization, and Prese[illegible]tion Best Practices
- Developing a Clean and Testable UI and Presentation Layers in Swift

- Validating UI Design and Getting Fast Feedback with UI Prototypes
- Working Effectively with Designers
- Efficiently Loading and Presenting Data on Screen
- Efficiently Loading Images in Reusable Cells
- Efficiently Prefetching Images when Cells are Near Visible
- Efficiently Canceling Data Loading Requests to Reduce Data Usage
- MVC: Implementing Best Practices and Variations in Swift
- Identifying and Fixing the Massive View Controller Anti-Pattern
- Multi-MVC Design: Breaking Down Complex Scenes into Tiny Collaborative MVCs
- MVVM: Implementing Best Practices and Variations in Swift
- Identifying and Fixing the Massive View Model Anti-Pattern
- Implementing Stateful and Stateless View Models in Swift
- MVP: Implementing Best Practices and Variations in Swift
- Identifying and Fixing the Massive Presenter Anti-Pattern
- Creating Reusable Cross-platform Presentation Layers with MVVM and MVP in Swift
- Test-driving MVC, MVVM, MVP, and their Variants
- Implementing and Testing Customer-Facing Localized Strings
- Inside-Out vs. Outside-In Development Strategies
- Safely Refactoring Code Backed by Automated Tests and Swift Types (Compiler!)
- Separating Platform-specific and Platform-agnostic Code using Swift Frameworks in Xcode
- Supporting Multiple Platforms on Swift Frameworks in Xcode
- Identifying and Solving Cyclic Dependencies (Retain Cycles) with the Proxy Design Pattern in Swift
- Adapter pattern Applied in Swift: Enabling components with incompatible interfaces to work together seamlessly
- Decorator Pattern Applied in Swift: Extending behavior of individual objects without changing their implementation
- Safely Adding Tests to Legacy Code
- Creating Reusable Components with Generics in Swift

What I've Learned in the Third Module (Persistence)

- Persistence Best Practices in Swift
- Developing a Testable Persistence Layer in Swift
- Choosing Between Persistence Options: In-Memory, UserDefaults, FileSystem, URLCache, Core Data, and other key persistence frameworks.
- Persisting and Retrieving data with URLCache in Swift
- Persisting and Retrieving data with Codable+FileSystem in Swift
- Persisting and Retrieving data with Core Data in Swift: modeling entities, contexts, concurrency model, and testing techniques
- Concurrency and Threading Best Practices in Swift
- Safely Managing Shared State in Multithreaded Environments
- Designing and Testing Thread-safe Components with DispatchQueue
- Dispath framework in Swift: Serial and Concurrent DispatchQueues, sync, async, barrier flags, and more
- Identifying, Debugging, and Solving Threading Race Conditions
- Thread Safety with Reference and Value Types in Swift
- Composite Reuse Principle (aka Prefer composition over inheritance) applied in Swift
- Protocol-Oriented Programming (POP) in Swift: Protocol inheritance, extensions, composition, and conditional conformance
- Protocol vs. Class inheritance in Swift
- Controlling the Current Date/Time and Other Environment Details During Tests
- Achieving a Healthy Distribution of Unit, Integration, End-To-End, and other testing strategies
- Speeding up Development with App Architecture to Develop Features in Parallel
- Speeding up Development with Iterative Design over Big Upfront Design
- Decoupling Business Logic From Infrastructure Details
- Breaking Down Monoliths into Modules: When, Why and How
- Entities vs. Value objects in Swift

- Data Transfer Objects (DTOs) to Decouple Modules in Swift
- Single Responsibility Principle (SRP) Applied in Swift
- Open-Closed Principle (OCP) Applied in Swift
- Liskov Substitution Principle (LSP) Applied in Swift
- Interface Segregation Principle (ISP) Applied in Swift
- Dependency Inversion Principle (DIP) Applied in Swift
- Don't Repeat Yourself (DRY) Principle Applied in Swift
- Command-Query Separation Principle Applied in Swift
- Functional Core / Imperative Shell Pattern Applied in Swift
- Performing Calendrical Calculations Correctly in Swift
- Domain Specific Languages (DSLs) Applied in Swift
- Achieving High Test Coverage with Triangulation
- Producing a Clean and Stable Codebase History in git
- Designing Side-Effect Free (Deterministic) Core Business Rules
- Maximizing Swift Code Correctness with Single Sources of Truth
- Integration Testing with Real Frameworks (Instead of Mocks)
- Measuring and Improving Test Times with xcodebuild
- Codebase Health Analysis Techniques
- Eliminating Hard-to-Read Nested Code (aka Arrow Code Anti-Pattern)
- Choosing Good Names in Swift

What I've Learned in the Second Module (Networking)

- Networking Best Practices in Swift
- URLSession, URLSessionDataTasks, URLProtocol, and the URL Loading System in Swift
- Effective Use of 3rd-party Networking Frameworks in Swift (Firebase, Alamofire,

etc.)

- Encoding/Decoding JSON API Data to Native Models Using Codable in Swift
- Developing a Testable Networking Layer in Swift
- Four Approaches to Test(-Drive) Network Requests in Swift: End-to-End, Subclass, Protocol-Based Mocking, and URLProtocol Stubbing
- What Many Apps Get Wrong About Reachability and How To Get It Right
- Speeding up Development using frameworks in Xcode
- Speeding up Development by Reducing Debugging Time in Xcode
- Automating a Continuous Integration (CI) Pipeline
- Object-Oriented Programming
- Functional Programming
- Test-Driven Development (TDD)
- Automated Tests
- Unit Testing
- Integration Testing
- End-to-end Testing
- Refactoring
- Version control with git
- Closures and Protocols as Abstractions in Swift
- Identifying, Debugging, and Solving Data Races in Xcode
- Pattern Matching in Swift
- Writing Safe Swift Code by Making Invalid Paths Unrepresentable
- Effective Dependency Management with Dependency Injection in Swift
- Effective Access Control in Swift
- Effective Error Handling in Swift
- Effective Memory Management in Swift
- Automating Memory Leak Detection in Swift
- Configuring Xcode Projects for Running Automated Tests
- Randomizing and Parallelizing Test Execution in Xcode
- Gathering, Understanding, and Improving Code Coverage in Xcode
- Testing Best Practices in Swift
- Common Test Doubles: Spy, Stub, Mock, Fake
- Testing Async Behavior in Swift
- Testing Error Cases in Swift
- Preventing Common Async Bugs in Swift

- Using Swift's Result and Error types

What I've Learned in the First Module (System Design)

- App Architecture Best Practices
- System Design and Requirements Analysis
- Thinking, Designing, and Drawing Diagrams like a Software Architect
- Dealing with Singletons and Globals: When, Why, How, and Better Alternatives
- SOLID Principles Applied in Practice in Swift
- Clean Architecture
- Modular Design
- Domain-Driven Design (DDD)
- Behavior-Driven Development (BDD)
- Use Case Analysis
- Domain Modeling
- Pair programming
- Effectively Developing Swift Apps Before the Backend/Design Is Ready

What I'll Learn Next

Click to see the full iOS Lead Essentials curriculum

© Essential Developer. All rights reserved.

Privacy Policy