UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE INGENIERÍA

Modernización de software heredado: Rediseño y desarrollo nativo de una app de salud en iOS

INFORME DE ACTIVIDADES PROFESIONALES

Que para obtener el título de

Ingeniero Mecatrónico

P R E S E N T A

Marcos Uriel Martinez Ortiz

ASESOR DE INFORME

M.A. Luis Yair Bautista Blanco

Ciudad Universitaria, Cd. Mx., 2025

PROTESTA UNIVERSITARIA DE INTEGRIDAD Y HONESTIDAD ACADÉMICA Y PROFESIONAL
(Titulación con trabajo escrito)

De conformidad con lo dispuesto en los artículos 87, fracción V, del Estatuto General, 68, primer párrafo, del Reglamento General de Estudios Universitarios y 26, fracción I, y 35 del Reglamento General de Exámenes, me comprometo en todo tiempo a honrar a la institución y a cumplir con los principios establecidos en el Código de Ética de la Universidad Nacional Autónoma de México, especialmente con los de integridad y honestidad académica.

De acuerdo con lo anterior, manifiesto que el trabajo escrito titulado MODERNIZACION DE SOFTWARE HEREDADO: REDISEÑO Y DESARROLLO NATIVO DE UNA APP DE SALUD EN IOS que presenté para obtener el titulo de INGENIERO MECATRÓNICO es original, de mi autoría y lo realicé con el rigor metodológico exigido por mi Entidad Académica, citando las fuentes de ideas, textos, imágenes, gráficos u otro tipo de obras empleadas para su desarrollo.

En consecuencia, acepto que la falta de cumplimiento de las disposiciones reglamentarias y normativas de la Universidad, en particular las ya referidas en el Código de Ética, llevará a la nulidad de los actos de carácter académico administrativo del proceso de titulación.

MARCOS URIEL MARTINEZ ORTIZ
Número de cuenta: 313090325

Resumen 1
Palabras clave 1
Abstract 2
Keywords 2
Introducción 3
Antecedentes 3
Planteamiento del problema 3
Objetivo general 3
Objetivos específicos 4
Metodología 4
Contenido 5
Capítulo 1 - Descripción del sistema 6
Ubicación geográfica 6
Ubicación sectorial 6
Comparativa sectorial en LATAM (2023) 6
Ubicación temporal 7
Dinámica de trabajo y estructura del proyecto 7
Antecedentes técnicos del sistema 8
Descripción del sistema 8
Capítulo 2 - Propuesta técnica y estrategia de reestructuración 10
Estado inicial del proyecto y diagnóstico técnico 10
Propuestas y cambios realizados en el sistema 11
Enfoque general de la solución 11
Propuesta: MVC mejorado 11
Estándares visuales y diseño modular 12
Integración segura y conectividad robusta 12
Pruebas, despliegue y mantenimiento 12

Planificación técnica y fases 13
Capacitación y fortalecimiento del equipo (versión final) 14
Capacitación técnica aplicada al desarrollo iOS 14
Fundamentos del lenguaje Swift moderno 14
Tipado seguro, opcionales y manejo de errores 14
Sintaxis concisa y expresiva 15
Inferencia de tipos 15
Organización y nomenclatura del código 16
Convenciones para nombrar variables y estructuras 16
Comentarios, claridad de intención y legibilidad 17
Organización de carpetas, archivos y grupos en Xcode 17
Arquitectura MVC con controladores auxiliares 18
Separación de responsabilidades en Model, View y Controller 18
Uso de ModelControllers y HelperControllers 19
Comparativa con MVVM y justificación de elección 19
Patrones de comunicación entre componentes 20
Delegados, closures, inyección de dependencias 20
Uso de Combine, tuplas y callbacks 22
Flujo de eventos y control de dependencias 23
Consumo de servicios REST y asincronía 24
Uso avanzado de URLSession 24
Comparativa entre URLSession y Alamofire 25
Programación con GCD y async/await 26
Gestión segura y eficiente del almacenamiento local 27
UserDefaults, Keychain, FileManager y Core Data 27
Estrategias para persistencia segura y sincronización 28
Diseño de interfaces en UIKit 29
Principios de diseño con Storyboards y Auto Layout 29
Patrones de navegación en UIKit 31
Control de versiones y flujo de trabajo con GitFlow 32

Tipos de ramas: main, develop, feature, release y hotfix 32

Buenas prácticas para mantenimiento, integración y revisión 34

Uso de etiquetas, CI/CD y pruebas automatizadas con XCTest 35

Capítulo 3 - Implementación práctica y validación funcional 36

Despliegue inicial del proyecto y aplicación de arquitectura modular 36

Punto de entrada y arquitectura de entorno en el inicio de sesión 39

Modularización de interfaces con Storyboard References 43

¿Qué es una Storyboard Reference? 43

Ventajas de usar Storyboard References 43

Ventajas frente al diseño 100% programático 43

Aplicación en el proyecto Wee 3.0 44

Implementación de múltiples métodos de almacenamiento local 45

Peticiones de red seguras y manejo de errores 47

Seguridad en la conexión con URLSession 47

Diseño genérico y controlado de respuestas de red 48

Controlador genérico con compatibilidad total con Codable 48

Uso de Result<T, Error> y propagación explícita de fallos 48

Modelo de error personalizado con semántica descriptiva 49

URLSession y su delegado para control avanzado de red 50

Separación de lógica con controladores auxiliares de errores 50

Integración de Compositional Layout y Diffable Data Source 51

¿Qué es Compositional Layout? 51

¿Qué es Diffable Data Source? 52

Aplicación en el proyecto Wee 3.0 52

Capítulo 4 – Cierre técnico y profesional del proyecto 55

Aplicación integral de conocimientos de ingeniería 55

Toma de decisiones tecnológicas y liderazgo técnico 55

Crecimiento profesional en entorno real 56

Condiciones del cierre del proyecto 57
Bibliografía **58**

Resumen

Durante una estancia profesional de seis meses en WeeCompany®, empresa del sector tecnológico especializada en Insurtech y salud digital, participé en el rediseño e implementación de una nueva aplicación móvil nativa para plataformas Apple. El problema principal que enfrentaba la organización era una aplicación existente con un alto grado de deuda técnica: ausencia de una arquitectura clara, código obsoleto, prácticas inconsistentes de comunicación entre vistas y una estructura de proyecto desorganizada, lo que dificultaba el desarrollo ágil de nuevas funcionalidades y comprometía la mantenibilidad del sistema.

Ante este panorama, mi labor se centró en diseñar y construir una nueva solución móvil desde cero, aplicando conocimientos actualizados de desarrollo nativo en iOS. Aunque no ocupé el rol formal de líder técnico, se me asignó la responsabilidad principal del proyecto. Entre mis funciones se incluyeron la creación del repositorio en Azure DevOps, el diseño de una arquitectura modular con separación de responsabilidades, la configuración de esquemas para distinguir entre ambientes de desarrollo y producción, la capacitación del equipo en prácticas modernas con UIKit y Storyboards, y la integración de herramientas como URLSession, Keychain y FileManager para la gestión segura de la información. También implementé flujos de trabajo con GitFlow y revisiones mediante Pull Requests para elevar el estándar de calidad del código.

El resultado fue una aplicación estructurada con una arquitectura limpia, fluida en su funcionamiento y preparada para múltiples plataformas (iOS, iPadOS y macOS mediante Catalyst). Se alcanzó un diseño adaptable a distintas resoluciones gracias a Auto Layout, y se optimizó la comunicación con el backend mediante una capa robusta de red y manejo centralizado de errores. Aunque mi participación concluyó antes del cierre definitivo del desarrollo, el avance entregado consolidó las bases de una solución moderna y sostenible que posiciona a la empresa para escalar sus servicios móviles con mayor eficiencia.

Palabras clave

Desarrollo nativo, Swift, SwiftUI, UIKit, arquitectura móvil, MVC, iOS, Xcode, Keychain, seguridad en aplicaciones móviles, Apple Developer, API REST, JSON, URLSession, patrón de diseño, re-ingeniería de software, refactorización, optimización de rendimiento, experiencia de usuario (UX), control de versiones, Git, integración continua, salud digital, aplicación médica, servicios de salud, videollamadas médicas, gestión de medicamentos, telemedicina, aseguradoras médicas, expediente clínico electrónico, bienestar emocional, salud mental, experiencia profesional, desarrollo de software, análisis de requerimientos, resolución de problemas técnicos, trabajo en equipo, liderazgo técnico, comunicación efectiva.

Abstract

During a six-month professional residency at WeeCompany®, a technology firm specialized in Insurtech and digital health, I participated in the redesign and implementation of a new native mobile application for Apple platforms. The main issue the organization faced was a legacy app with a high level of technical debt: lack of a clear architecture, obsolete code, inconsistent view-to-view communication practices, and a disorganized project structure, which hindered the agile development of new features and compromised the system's maintainability.

Given this scenario, my work focused on designing and building a new mobile solution from scratch, applying up-to-date knowledge of native iOS development. Although I did not formally hold the role of technical lead, I was assigned the primary responsibility for the project. My tasks included setting up the repository in Azure DevOps, designing a modular architecture with clear separation of concerns, configuring schemes to distinguish between development and production environments, training the team in modern practices using UIKit and Storyboards, and integrating tools such as URLSession, Keychain, and FileManager for secure data management. I also implemented GitFlow workflows and code review processes through Pull Requests to raise the overall code quality.

The result was a well-structured application with a clean architecture, fluid performance, and readiness for deployment across multiple platforms (iOS, iPadOS, and macOS via Catalyst). The interface was designed to be responsive across various screen sizes using Auto Layout, and the backend communication was optimized through a robust networking layer and centralized error handling. Although my participation concluded before the final release, the work delivered established a solid foundation for a modern and sustainable solution, enabling the company to scale its mobile services more efficiently.

Keywords

Native development, Swift, SwiftUI, UIKit, mobile architecture, MVC, iOS, Xcode, Keychain, mobile app security, Apple Developer, REST API, JSON, URLSession, design patterns, software re-engineering, refactoring, performance optimization, user experience (UX), version control, Git, continuous integration, digital health, medical application, healthcare services, medical video calls, medication management, telemedicine, health insurance providers, electronic health records, emotional wellness, mental health, professional experience, software development, requirements analysis, technical problem solving, teamwork, technical leadership, effective communication.

Introducción

Antecedentes

En el contexto de la transformación digital en el sector salud, WeeCompany® se posiciona como una empresa líder en Latinoamérica en el ámbito Insurtech, con un enfoque en el desarrollo de soluciones tecnológicas para la gestión de servicios médicos y seguros de gastos médicos y vida. Fundada en 2016, su objetivo principal es ofrecer una plataforma que interconecte a aseguradoras, financiadoras y proveedores de salud (hospitales, farmacias, laboratorios, clínicas, etc.) mediante herramientas digitales que mejoren la eficiencia operativa y la experiencia del usuario.

Al momento de mi incorporación a la empresa, el equipo contaba con una aplicación móvil funcional pero altamente limitada en términos de arquitectura, mantenibilidad y escalabilidad. El proyecto presentaba una profunda deuda técnica y una estructura organizativa inconsistente, lo que comprometía su capacidad para incorporar nuevas funcionalidades y para posicionarse como un producto de tipo software as a service (SaaS). Esta situación representó una oportunidad para intervenir desde una perspectiva de re-ingeniería, aplicando conocimientos actualizados del ecosistema Apple para el rediseño completo del producto móvil.

Planteamiento del problema

El estado inicial de la aplicación incluía una mezcla incorrecta de patrones arquitectónicos (MVP y MVVM), uso programático de UIKit sin aprovechamiento de Storyboards ni Auto Layout, comunicación inconsistente entre vistas, y un sistema de dependencias basado en herramientas obsoletas como CocoaPods. Además, existía un desconocimiento generalizado dentro del equipo sobre las buenas prácticas de desarrollo nativo, arquitectura de software móvil, y lineamientos de diseño de Apple, lo cual limitaba su capacidad de evolución. A nivel organizacional, tampoco existía un sistema de versionado ni un control efectivo de calidad sobre el código.

Objetivo general

Desarrollar una nueva versión de la aplicación móvil de WeeCompany®, denominada Wee 3.0, que resolviera los problemas técnicos existentes y permitiera su escalabilidad, adaptabilidad y comercialización como un software personalizable tipo software as a service para distintos clientes.

Objetivos específicos

- Reestructurar la arquitectura de la aplicación bajo principios modernos de modularidad y separación de responsabilidades.
- Capacitar al equipo de desarrollo en el uso actualizado de UIKit, SwiftUI, y diseño multiplataforma para iOS, iPadOS y macOS.
- Implementar una capa de comunicación de red robusta, concurrente y segura, manejo de errores centralizado y control de estados.
- Capacitar al equipo de UI/UX en los lineamientos de diseño nativo de Apple (Apple Human Interface Guidelines) para mejorar la experiencia de usuario.
- Establecer un sistema de versionado y revisión de código con GitFlow y Pull Requests para asegurar la calidad técnica del desarrollo.
- Desarrollar estrategias de solución ágil de problemas en entornos productivos.

Metodología

El desarrollo del proyecto se realizó bajo un enfoque ágil, concretamente mediante la metodología Scrum, con el acompañamiento de un Scrum Master y una Product Owner (PO). El trabajo se organizó en sprints de dos semanas, durante los cuales se asignaban historias de usuario desde un backlog centralizado. Estas historias eran discutidas, puntuadas y refinadas por el equipo, y cada desarrollador estimaba el esfuerzo requerido y registraba las horas destinadas a su desarrollo. Las decisiones técnicas, dificultades y avances se discutían en sesiones de planeación y retrospectiva.

El proceso fue completamente iterativo, con validación continua mediante revisiones de código, pruebas funcionales y entregas parciales orientadas a negocio. Las tareas y métricas fueron gestionadas desde la plataforma Azure DevOps, la cual permitía un seguimiento puntual del cumplimiento de los objetivos planteados en cada sprint.

Contenido

Este trabajo se estructura en tres capítulos, además de la introducción, el resumen, las conclusiones, referencias y anexos.

- **Capítulo 1 – Descripción del sistema**
    - Se presenta el entorno técnico y organizacional previo al rediseño de la aplicación. Se describe el sistema heredado en producción, incluyendo su estructura, problemáticas, deuda técnica acumulada y deficiencias arquitectónicas. Se contextualiza también mi incorporación al equipo, el organigrama de la empresa y las responsabilidades asignadas.
- **Capítulo 2 – Propuesta técnica y estrategia de reestructuración**
    - Se detalla el análisis realizado y las decisiones técnicas adoptadas para desarrollar una nueva versión de la aplicación. Se expone la arquitectura seleccionada, las herramientas utilizadas, la estructura modular, los principios de seguridad y pruebas, así como el enfoque formativo aplicado para estandarizar conocimientos dentro del equipo de desarrollo.
- **Capítulo 3 – Implementación práctica y validación funcional**
    - Se documenta el desarrollo efectivo del sistema, ilustrado mediante evidencia visual y análisis del comportamiento de la aplicación en diferentes dispositivos y flujos. Se explican los resultados obtenidos a nivel de código, estructura, experiencia de usuario, navegación y adaptabilidad multiplataforma. Esta sección integra la visión técnica con los resultados tangibles alcanzados durante el proceso.
- **Capítulo 4 – Cierre técnico y profesional del proyecto**
    - Se reflexiona sobre el aprendizaje adquirido a lo largo del proyecto, el rol asumido en la toma de decisiones, y la aplicación integral de conocimientos de ingeniería. Además, se describe el estado en que se entregó el sistema, la continuidad propuesta para su evolución y el impacto profesional que esta experiencia representa en el desarrollo de una carrera en software dentro del campo de la ingeniería mecatrónica.

Capítulo 1 - Descripción del sistema

Ubicación geográfica

El proyecto Wee 3.0 se desarrolló en la Ciudad de México, en modalidad 100 % presencial, dentro de las oficinas corporativas de WeeCompany®. Esta ubicación representa un punto estratégico dentro del país, ya que la capital concentra a las principales sedes de aseguradoras, instituciones financieras, hospitales privados y empresas tecnológicas que lideran la transformación digital en sus respectivos sectores.

El entorno profesional en la CDMX es altamente competitivo; existe una abundante oferta de talento técnico y una exigencia constante por la excelencia profesional. Este contexto favoreció una cultura organizacional orientada al rendimiento, la innovación continua y el trabajo colaborativo. Dentro de este ecosistema, WeeCompany® se posiciona como un actor con una visión clara: simplificar la relación entre pacientes, prestadores de salud y aseguradoras mediante soluciones digitales integrales (WeeCompany, s.f.).

Ubicación sectorial

Fundada en 2016, WeeCompany® es reconocida como la primera empresa InsurTech en América Latina con un enfoque centrado específicamente en el sector salud. Su misión es mejorar la experiencia de todos los actores involucrados en el sistema médico y asegurador, con énfasis en el bienestar del paciente. Su visión es consolidarse como el sistema operativo líder para la gestión del sector salud y seguros en LATAM (WeeCompany, s.f.).

Para ello, ha construido un ecosistema digital robusto que conecta aseguradoras, financiadoras de servicios médicos y una amplia red de proveedores de salud, incluyendo farmacias, laboratorios, hospitales, clínicas, consultorios y profesionales independientes. A través de plataformas digitales automatizadas, busca optimizar procesos como la gestión de reclamos, la administración de casos clínicos, el control de gastos, la prescripción médica digital y la atención remota por telemedicina (WeeCompany, s.f.).

Comparativa sectorial en LATAM (2023)

Durante 2023, el ecosistema InsurTech y HealthTech en América Latina experimentó un crecimiento acelerado. México se posicionó como el segundo mercado más grande de la región en insurtech, solo detrás de Brasil, con más de 120 startups activas en este sector (MAPFRE, 2024). Además, el mercado

healthtech mexicano alcanzó un valor estimado de 1,930 millones de dólares, siendo uno de los ecosistemas más dinámicos y diversificados de la región, con más de 100 empresas clasificadas en áreas como telemedicina, prescripción digital y gestión de enfermedades crónicas (ICEX, 2024).

En este contexto, el enfoque de WeeCompany® sobresale por su carácter modular, integrado y centrado en el usuario final. A diferencia de startups como Betterfly (Chile), que combina seguros de vida con hábitos saludables enfocados en beneficios corporativos (Endeavor Hub, 2022), o Diagnostikare (México), centrada en atención médica primaria virtual (Diagnostikare, s.f.), WeeCompany® articula todo un ciclo de valor que va desde la contratación de seguros, la validación de pólizas, la consulta médica y el seguimiento emocional del paciente, hasta la gestión de pagos, documentos y trámites administrativos con aseguradoras. Esta integración vertical le otorga una ventaja competitiva única en la región.

Ubicación temporal

Mi estancia profesional en WeeCompany® se desarrolló entre el 28 de febrero y septiembre/octubre de 2024, aunque el proyecto Wee 3.0 fue conceptualizado y planificado en 2023, durante una etapa crítica de evaluación interna. En ese año, la compañía enfrentaba una situación de estancamiento técnico con su aplicación anterior (Wee 2.0), la cual no cumplía con los estándares de estabilidad, escalabilidad ni usabilidad necesarios para su evolución como producto comercial.

Este contexto coincidió con la necesidad de consolidar un producto digital tipo Software as a Service (SaaS) que pudiera ser personalizado por aseguradoras y licenciado como solución integral para gestionar sus usuarios, coberturas, gastos y servicios. La presión de contratos vigentes, clientes con requerimientos específicos y objetivos de expansión regional generaron un entorno donde era indispensable acelerar el rediseño de la aplicación.

Durante este periodo, se adoptaron prácticas ágiles mediante el uso de Scrum, se establecieron sprints de 15 días para gestionar la entrega continua de valor, y se programaron sesiones trimestrales de demo con stakeholders de negocio. Estas condiciones temporales definieron el ritmo del desarrollo y marcaron el inicio de un proceso de transformación estructural a nivel técnico y organizacional dentro de la empresa.

Dinámica de trabajo y estructura del proyecto

El desarrollo del proyecto se llevó a cabo dentro de un entorno colaborativo bien estructurado, que combinaba herramientas tecnológicas modernas con una metodología ágil orientada a resultados. Para la implementación del sistema se contó con el equipo necesario, incluyendo dispositivos Apple como una MacBook Pro M1, varios modelos de iPhone, iPad y Apple Watch, todos esenciales para asegurar compatibilidad y rendimiento multiplataforma. A nivel organizativo, se utilizó la suite Microsoft 365, con Azure DevOps como plataforma central de coordinación. Desde ahí se gestionaban los repositorios de código, el control de versiones, los tableros de tareas y la documentación técnica; mientras que herramientas como Microsoft Teams y OneDrive facilitaban la comunicación y el trabajo compartido.

El equipo adoptó la metodología Scrum, con sprints quincenales que iniciaban con sesiones de planeación lideradas por la Product Owner y el Scrum Master. Durante estas reuniones se revisaban las historias de usuario, se discutía su alcance y se estimaban los esfuerzos técnicos requeridos. Las tareas se dividían en subtareas específicas en el tablero de Azure y se realizaba un seguimiento constante de su avance. Una vez completadas, se realizaban revisiones cruzadas mediante pull requests, promoviendo un entorno horizontal, sin jerarquías técnicas impuestas, donde cada desarrollador podía aportar a la mejora del código.

Las entregas se hacían de forma iterativa: cada quince días se distribuían versiones menores mediante Firebase, dirigidas principalmente a validaciones internas de producto; mientras que cada trimestre se generaban versiones mayores que se presentaban al área de negocio, con funcionalidades más robustas, documentación completa y pruebas visuales en todos los dispositivos objetivo. En este proceso, la Product Owner validaba el cumplimiento funcional y el Scrum Master velaba por la eficiencia del equipo y el equilibrio en la carga de trabajo.

El equipo técnico estaba conformado por desarrolladores iOS y Android, mientras que las decisiones de producto y prioridades eran coordinadas por la PO. Esta estructura permitió mantener una dinámica ágil, iterativa y medible, con resultados funcionales tangibles al cierre de cada sprint.

Antecedentes técnicos del sistema

Antes del inicio del proyecto Wee 3.0, la empresa operaba con una versión anterior de su aplicación móvil, conocida como Wee 2.0, la cual presentaba múltiples limitaciones que comprometían su rendimiento, estabilidad y escalabilidad. Esta versión acumulaba una importante deuda técnica producto de decisiones arquitectónicas inconsistentes, falta de documentación y una estructura de código difícil de mantener.

Durante mis primeras semanas en la empresa, se me asignó la tarea de realizar un diagnóstico del sistema en producción. El análisis evidenció deficiencias significativas tanto en la aplicación móvil como en los servicios backend que la soportaban. Estas observaciones iniciales fueron presentadas al equipo de desarrollo y dirección técnica, sirviendo como base para definir la necesidad de rediseñar la plataforma desde sus fundamentos, lo cual dio origen al proyecto Wee 3.0, abordado en detalle en el siguiente capítulo.

Descripción del sistema

La aplicación móvil desarrollada por WeeCompany®, en su versión anterior (Wee 2.0), constituía una solución digital integral dirigida principalmente a usuarios con pólizas de seguro médico, aunque también ofrecía funcionalidades abiertas al público general a través de un modelo basado en membresías. Su objetivo era centralizar y facilitar el acceso a diversos servicios de salud y de seguros, mediante una plataforma tecnológica que automatizara procesos que tradicionalmente eran fragmentados, lentos o dependientes de múltiples intermediarios.

Las funcionalidades principales de la aplicación incluían la consulta médica en línea, la localización de especialistas por geolocalización, la gestión y visualización de pólizas, el surtido digital de recetas médicas y la recepción de notificaciones o recordatorios de salud. Estas acciones se complementaban con servicios auxiliares como el registro de estado emocional mediante un diario personal, el acceso a una biblioteca de videos educativos sobre salud y un asistente conversacional basado en inteligencia artificial (ChatGPT), orientado a resolver dudas generales de salud.

La app estaba diseñada para cubrir tanto necesidades básicas como complejas del usuario asegurado. Por ejemplo, permitía solicitar atención médica inmediata o programada, registrar su póliza de seguro, subir documentación en caso de requerimientos administrativos y consultar su historial médico. Esta integración simplificada de servicios hacía que el usuario solo tuviera que interactuar con la app móvil, mientras que WeeCompany gestionaba internamente toda la operación con las aseguradoras, médicos y farmacias afiliadas.

En el plano organizativo, la aplicación se dividía en módulos internos conectados a través de servicios REST, comunicándose exclusivamente con un backend privado desarrollado por la misma empresa. La interacción con aseguradoras y prestadores de servicios no ocurría directamente desde la app, sino a través de una plataforma web empresarial donde dichas entidades gestionaban la información de sus clientes y configuraban sus permisos. Esta arquitectura centralizada aseguraba que la app operara como un nodo cliente, accediendo a recursos ya preprocesados por el entorno administrativo corporativo.

Cabe destacar que muchas de las funcionalidades avanzadas, como las consultas médicas virtuales, la emisión de recetas, y la conexión con la red médica profesional, eran provistas por una empresa hermana: Wee Medic. Aunque operativamente separadas, ambas plataformas estaban integradas a nivel funcional, lo que permitía ofrecer una experiencia fluida y coherente al usuario final.

Gracias a su enfoque holístico, Wee 2.0 se convirtió en una de las aplicaciones más completas en el mercado mexicano en términos de servicios médicos móviles. Su principal valor diferencial radicaba en su capacidad de automatizar procesos médicos y administrativos complejos, permitiendo al usuario gestionar integralmente su salud desde una sola interfaz digital.

Capítulo 2 - Propuesta técnica y estrategia de reestructuración

Estado inicial del proyecto y diagnóstico técnico

Al momento de mi incorporación a WeeCompany®, la aplicación en uso era conocida como Wee 2.0, aunque internamente existían versiones incrementales como 2.5.4, sin cambios funcionales sustanciales. Esta versión aún estaba en producción, lo que representaba una dificultad adicional: cualquier modificación o corrección debía realizarse sin comprometer los servicios activos para usuarios reales, ya que existían compromisos contractuales con aseguradoras y clientes empresariales.

Durante mi primer sprint, se me asignó como prioridad la elaboración de un diagnóstico técnico detallado, que permitiera evaluar la viabilidad de continuar con la app actual o iniciar un rediseño total. El análisis incluyó una revisión del código fuente, la estructura arquitectónica, los flujos de navegación, las herramientas utilizadas y la experiencia de usuario.

Los hallazgos fueron contundentes: la aplicación sufría una acumulación crítica de deuda técnica, carecía de una arquitectura clara y utilizaba patrones híbridos mal implementados, como una mezcla fallida entre MVP y MVVM. El proyecto presentaba una organización desordenada en sus carpetas, archivos y módulos. Se identificó código obsoleto, sin uso, duplicado y sin pruebas unitarias, lo cual hacía extremadamente costosa cualquier intervención o ampliación funcional.

La integración de dependencias se realizaba mediante CocoaPods, sin control de versiones, y en algunos casos con librerías sin mantenimiento. La comunicación entre componentes carecía de un patrón unificado: en distintos puntos del proyecto se utilizaban closures, RxSwift o incluso acceso directo a objetos sin encapsulación, lo que generaba altísimo acoplamiento y rompía el principio de separación de responsabilidades.

En cuanto al backend, se observó una situación similar: muchos servicios carecían de documentación, no existía una colección actualizada en Postman ni una especificación clara en Swagger. Las respuestas del servidor eran poco robustas, con códigos HTTP 200 incluso en operaciones fallidas, y la detección de errores dependía de la interpretación de textos embebidos en campos anidados del JSON. Esto imposibilitaba establecer una política clara y predecible de manejo de errores en la app cliente.

El entorno de desarrollo tampoco contemplaba estructuras diferenciadas para ambientes de pruebas, desarrollo o producción. No existía un sistema de control de calidad automatizado ni lineamientos compartidos entre el equipo de desarrollo. La falta de documentación interna agravaba esta situación, generando una dependencia excesiva del conocimiento tácito entre los miembros del equipo.

Ante esta situación, y considerando las necesidades urgentes de escalabilidad y estabilidad del producto, propuse formalmente la construcción de una nueva versión de la aplicación, bajo una estructura arquitectónica moderna, clara y adaptable, con énfasis en: modularidad, personalización por cliente, experiencia de usuario coherente con las Human Interface Guidelines, uso eficiente de tecnologías nativas de Apple y buenas prácticas de ingeniería de software.

Propuestas y cambios realizados en el sistema

A partir del diagnóstico inicial, y en alineación con los objetivos estratégicos de WeeCompany®, propuse el desarrollo de una nueva versión de la aplicación, estructurada como un producto mínimo viable (MVP) que sentara las bases de una plataforma escalable, personalizable y sostenible a largo plazo. Este nuevo sistema, denominado Wee 3.0, debía resolver no solo las deficiencias técnicas de Wee 2.0, sino también responder a las expectativas del mercado insurtech y healthtech con una solución flexible adaptable a distintos clientes y aseguradoras.

Enfoque general de la solución

El nuevo diseño partía de la creación de un MVP con una arquitectura documentada y modular, que permitiera incorporar o remover funcionalidades según el contrato o configuración de cada cliente. Esto implicaba separar la lógica por módulos funcionales reutilizables, definir estilos visuales como clases y estructuras en Swift, y asegurar una navegación fluida mediante estructuras compatibles con UIKit, pero abiertas a migraciones progresivas hacia SwiftUI.

Propuesta: MVC mejorado

Aunque inicialmente se consideró adoptar el patrón MVVM para la nueva aplicación Wee 3.0, tras una evaluación profunda de las características del proyecto, la madurez del equipo y los objetivos de corto y mediano plazo, se optó por implementar una versión mejorada del patrón Model-View-Controller (MVC).

Esta decisión se alineó con las prácticas recomendadas por Apple para aplicaciones basadas en UIKit, y respondió a tres factores clave:

- La estructura nativa de UIKit está optimizada para MVC, lo que reduce fricciones y facilita la integración con herramientas del ecosistema Apple.
- El equipo contaba con mayor experiencia en este patrón, lo que garantizaba una curva de adopción más rápida y una mejor colaboración entre miembros con distintos niveles de experiencia.
- La implementación de ModelControllers y HelperControllers permitió modularizar el código y distribuir responsabilidades, evitando el problema común del Massive View Controller.

El uso de ModelControllers permitió aislar la lógica de negocio y operaciones complejas fuera de los controladores de vista, mientras que los HelperControllers facilitaron la consolidación de servicios auxiliares como red, almacenamiento seguro y utilerías compartidas. Esta adaptación hizo posible mantener los principios de separación de responsabilidades y reusabilidad propios de MVVM, sin comprometer la simplicidad y fluidez operativa de MVC.

Esta decisión no solo facilitó el mantenimiento y la escalabilidad del proyecto, sino que también mejoró su capacidad de prueba, al permitir aislar unidades funcionales independientes, como el manejo de datos o servicios de red, con una estructura clara y controlada.

Estándares visuales y diseño modular

Se definieron nuevos lineamientos de diseño centrados en el cumplimiento estricto de las Human Interface Guidelines de Apple, incluyendo adaptabilidad con Auto Layout y soporte para diferentes tamaños de pantalla. La personalización visual se logró mediante estructuras y clases que centralizaban estilos, colores, fuentes y componentes visuales. Se utilizó Interface Builder con storyboard references para mantener independencia de vistas y reducir el acoplamiento visual.

Integración segura y conectividad robusta

La comunicación con servicios backend se implementó usando URLSession, con manejo avanzado de estados y errores. Para garantizar la protección de los datos del usuario, se incorporaron prácticas de cifrado modernas utilizando Keychain y CryptoKit, así como las políticas recomendadas por Apple mediante App Transport Security. Esta estructura permitió asegurar los datos en tránsito y en reposo, cumpliendo con estándares de seguridad de nivel empresarial.

Pruebas, despliegue y mantenimiento

Desde las primeras fases del proyecto se reconoció la necesidad de contar con una estrategia formal de pruebas y despliegue continuo. Sin embargo, a diferencia de lo inicialmente planteado, no se contó con un equipo DevOps especializado, por lo que los propios desarrolladores asumimos la responsabilidad de investigar, implementar e integrar los mecanismos necesarios dentro del ecosistema de Azure DevOps.

El proceso de aprendizaje incluyó la exploración de herramientas como pipelines de Azure, gestión de ramas con políticas de revisión, automatización de compilaciones y pruebas básicas antes de cada entrega. Esta fase de formación fue parte del proceso de capacitación técnica transversal, y se abordó colaborativamente entre los miembros del equipo, en sesiones conjuntas y pruebas iterativas de implementación.

Las validaciones funcionales se realizaban al cierre de cada sprint quincenal, bajo el marco de trabajo Scrum, utilizando un sistema de versiones distribuidas mediante Firebase para pruebas internas. La validación de historias de usuario quedaba a cargo de la Product Owner, mientras que entre desarrolladores se mantenía una política de revisión cruzada del código (pull requests) como mecanismo informal de control de calidad.

Si bien el pipeline de CI/CD aún estaba en construcción al finalizar mi estancia, se sentaron las bases para una implementación futura sostenible, con principios claros de control de versiones, automatización y revisión continua

Planificación técnica y fases

El desarrollo de Wee 3.0 fue concebido como un proyecto completamente independiente, iniciado desde cero, sin aprovechar componentes del código heredado debido al alto grado de deuda técnica e incompatibilidad estructural detectado en Wee 2.0. Esta decisión se tomó tras una auditoría inicial que reveló que ningún módulo del sistema anterior era reutilizable sin comprometer los estándares de calidad, rendimiento y seguridad que se buscaban establecer.

El proyecto se diseñó para ejecutarse de manera paralela al mantenimiento de la versión anterior, la cual debía permanecer activa y funcional para cumplir con contratos vigentes y compromisos con aseguradoras ya firmados. Esta doble exigencia obligó a una división estratégica del equipo: por un lado, garantizar la estabilidad mínima de Wee 2.0; por otro, avanzar con el diseño, desarrollo y validación progresiva de Wee 3.0.

La planificación técnica se estructuró en cinco fases:

1. Separación operativa de ambientes para evitar interferencias entre versiones.
2. Definición de arquitectura base y creación de módulos funcionales desde cero, organizados bajo el patrón MVC mejorado.
3. Establecimiento de estándares de codificación y control de versiones, con ramas separadas para desarrollo, pruebas y producción.
4. Implementación iterativa de funcionalidades en sprints, priorizando componentes reutilizables y aislados.
5. Preparación para la transición gradual, diseñando rutas de sustitución funcional sin interrumpir servicios críticos.

El cronograma original estimaba entre 6 y 8 meses de desarrollo activo para una versión estable del MVP, sujeto a ajustes según la disponibilidad de personal, avances en capacitación y soporte requerido por la versión anterior.

Capacitación y fortalecimiento del equipo (versión final)

Uno de los componentes más importantes del proyecto Wee 3.0 fue la formación técnica interna, impulsada por la necesidad de estandarizar criterios de desarrollo, elevar el nivel técnico del equipo, y facilitar la adopción de nuevas prácticas orientadas a la calidad y seguridad del software. Esta responsabilidad fue asumida directamente por mí como parte de mi participación activa en el proyecto.

Se diseñó y entregó un programa de capacitación intensiva, estructurado en torno a las buenas prácticas del desarrollo iOS moderno, centrado en el lenguaje Swift, el uso adecuado de UIKit y su arquitectura MVC, y herramientas esenciales como URLSession, Keychain, GCD y GitFlow. También se abordaron patrones de comunicación en Swift (delegados, closures, inyección de dependencias), diseño de flujos con navegación programática, estructura modular y consumo seguro de servicios backend.

A diferencia de una capacitación teórica, el enfoque fue aplicado y práctico, integrando cada contenido en tareas reales dentro del ciclo de desarrollo. Esto incluyó documentación, presentaciones, revisión de código, ejemplos funcionales y resolución colaborativa de problemas. La retroalimentación fue continua y la transferencia de conocimiento quedó asentada como parte del valor técnico duradero del proyecto.

Capacitación técnica aplicada al desarrollo iOS

Esta sección profundizará en los contenidos específicos impartidos al equipo, incluyendo fundamentos de Swift, patrones de diseño, control de versiones, seguridad, asincronía y principios de arquitectura limpia adaptados a MVC.

Fundamentos del lenguaje Swift moderno

Tipado seguro, opcionales y manejo de errores

Uno de los elementos clave de la formación impartida al equipo fue la comprensión del modelo de seguridad de tipos que define la base del lenguaje Swift. A diferencia de lenguajes más flexibles pero propensos a errores en tiempo de ejecución, Swift obliga a que cada variable, constante y expresión tenga un tipo bien definido. Esta característica reduce sustancialmente el margen de error y permite al compilador detectar incongruencias en tiempo de compilación, favoreciendo un desarrollo más confiable y mantenible.

El concepto de opcionalidad en Swift se abordó como una de las principales diferencias frente a otros lenguajes. Los opcionales permiten representar explícitamente la posibilidad de que un valor esté ausente, usando el tipo T?. Se trabajaron técnicas de desempaquetado seguro mediante if let y guard let, el uso de optional chaining para evitar cascadas de validaciones innecesarias, y el operador ?? como forma clara de definir valores predeterminados. También se explicó detalladamente por qué el uso de ! (force unwrap)

debe considerarse una mala práctica en producción, dado que puede conducir a fallos severos en tiempo de ejecución si el valor resulta ser nil.

Complementariamente, se introdujo el sistema de manejo de errores de Swift, basado en el protocolo Error y las instrucciones do, try, catch. Se discutió cómo Swift obliga al desarrollador a declarar qué funciones pueden lanzar errores y a tratarlos explícitamente, reforzando el principio de responsabilidad y previsión. Esta combinación entre opcionales y manejo de errores estructurado refuerza la filosofía de seguridad y claridad que caracteriza a Swift como lenguaje moderno.

Sintaxis concisa y expresiva

Swift fue diseñado con un objetivo claro: facilitar la escritura de código que sea expresivo, legible y cercano al lenguaje natural. Esta cualidad fue una de las que más se enfatizó durante la capacitación, particularmente para ayudar al equipo a escribir código más limpio, comprensible y mantenible.

Se analizaron ejemplos concretos de cómo la sintaxis clara de Swift permite evitar redundancias, reducir el uso de estructuras verbosas y construir funciones que se explican por sí mismas. Un caso ilustrativo fue el uso de etiquetas de parámetros en funciones (sayHello(to:and:)), lo que mejora radicalmente la comprensión del propósito de cada argumento sin necesidad de documentación adicional.

También se practicó el uso de closures en línea, la estructura compacta de expresiones como map, filter, reduce, y el uso elegante de inicializadores, métodos encadenados y control de flujo simplificado con guard. Todos estos elementos contribuyen a construir un código que no solo es más rápido de escribir, sino también más fácil de leer y revisar, lo cual es vital en proyectos colaborativos de mediano y largo plazo.

Esta claridad estructural fue reforzada como parte de una práctica transversal: escribir código para humanos, no solo para la máquina, asegurando que cualquier miembro del equipo pudiera leer, comprender y continuar trabajando sobre una base común, sin necesidad de depender del contexto tácito del autor original del código.

Inferencia de tipos

La inferencia de tipos es una de las características que distingue a Swift de otros lenguajes de programación fuertemente tipados. Gracias a esta funcionalidad, el compilador puede deducir el tipo de una variable a partir del valor que se le asigna, permitiendo escribir menos código sin comprometer la seguridad del sistema de tipos.

Durante la capacitación, se explicó cómo esta funcionalidad mejora la fluidez del desarrollo, haciendo que el código sea más conciso, sin necesidad de repetir tipos evidentes (let edad = 28 infiere que edad es un Int). Sin embargo, se hizo énfasis en que esta facilidad no debe llevar al abuso: en contextos donde la semántica del tipo no es evidente, es preferible especificar el tipo explícitamente, por claridad y para evitar errores sutiles.

Un ejemplo práctico abordado fue el caso de opcionales declarados sin valor inicial, donde la inferencia no es capaz de determinar el tipo sin una anotación explícita (var mensaje: String?). Este caso sirvió para ilustrar cómo la inferencia tiene límites y cómo el desarrollador debe complementar al compilador cuando el contexto no es suficiente.

El equipo aprendió a combinar la inferencia con la declaración explícita como una herramienta de equilibrio: usarla para acelerar el desarrollo cuando el tipo es evidente y recurrir a anotaciones claras cuando el tipo o su propósito no lo son. Esta estrategia no solo mantiene la legibilidad del código, sino que potencia el sistema de tipos de Swift como una forma activa de prevenir errores y comunicar intención.

Organización y nomenclatura del código

Convenciones para nombrar variables y estructuras

Durante la capacitación, se enfatizó la importancia de seguir las convenciones de nomenclatura establecidas por Swift para garantizar la coherencia, escalabilidad y legibilidad del código. Estas convenciones, alineadas con las Swift API Design Guidelines (Apple, s.f.-a), son fundamentales para mantener un código profesional y facilitar su mantenimiento colaborativo en equipos multidisciplinarios.

Las principales convenciones abordadas incluyeron:

- Tipos (clases, estructuras, enumeraciones y protocolos): Uso de UpperCamelCase, es decir, cada palabra comienza con mayúscula y no se emplean guiones bajos. Ejemplos: UserProfile, NetworkManager, DataParser.

- Variables y constantes: Uso de lowerCamelCase, iniciando con minúscula y capitalizando las palabras subsiguientes. Ejemplos: userName, maxRetries, apiEndpoint.

- Funciones y métodos: Mismo formato lowerCamelCase, con nombres que describan con claridad la acción que realizan. Ejemplos: calculateScore(), fetchData(), updateUserProfile().

- Enumeraciones (casos): También en lowerCamelCase, asegurando que cada caso sea claro y representativo del estado. Ejemplo:

- Parámetros genéricos: Letras mayúsculas simples como T, U, V, o nombres en UpperCamelCase para mayor expresividad:

```
struct Stack<Element> {
    var items: [Element] = []
}
```

Figura 1. Definición de una estructura genérica Stack en Swift.

También se abordó el uso de prefijos semánticos para variables booleanas como is, has, o should

(isUserLoggedIn, hasAccess, shouldDisplayAlert) para mejorar la semántica del código (Apple, s.f.-a). Estas convenciones ayudan a que el compilador proporcione asistencia contextual, a que el código sea auto-explicativo, y a facilitar la integración con herramientas de análisis estático.

Comentarios, claridad de intención y legibilidad

La documentación interna del código es una herramienta crítica para mantener la claridad, robustez y escalabilidad de una base de código compartida. Durante la formación, se abordaron las buenas prácticas para documentar código en Swift, con énfasis en el uso de Swift Markup, un conjunto de convenciones de marcado adoptadas por Apple para generar documentación directamente desde el código fuente (Apple, 2023a; Apple, 2021a).

Se enseñó a utilizar tanto comentarios informativos (//, /* */) como comentarios de documentación (///, /** */), que permiten a Xcode mostrar documentación estructurada mediante Quick Help.

Ejemplo:

```
/// Calcula el área de un rectángulo.
/// - Parameters:
///   - width: Ancho del rectángulo.
///   - height: Altura del rectángulo.
/// - Returns: El área calculada.
func calculateArea(width: Double, height: Double) -> Double {
    return width * height
}
```

Figura 2. Función calculateArea documentada con comentarios estructurados en Swift para calcular el área de un rectángulo.

Además, se introdujo la herramienta DocC, lanzada por Apple para permitir a los desarrolladores generar documentación de sus proyectos de forma automática, incluyendo encabezados, listas, ejemplos, imágenes y enlaces interactivos (Apple, 2021a; Apple, 2021b). Se explicó cómo estructurar los comentarios para integrarse con DocC y generar documentación navegable en Xcode o exportarla como documentación web.

Este enfoque favorece el desarrollo de sistemas sostenibles, donde el conocimiento se transmite dentro del propio código y no solo mediante documentación externa. También mejora la comunicación entre equipos, el onboarding de nuevos desarrolladores y la transparencia del diseño técnico.

Organización de carpetas, archivos y grupos en Xcode

Una estructura de proyecto clara es indispensable para cualquier equipo de desarrollo. Incluso aplicaciones de mediana escala pueden acumular decenas o cientos de archivos. Por ello, durante la capacitación se abordaron las mejores prácticas para organizar un proyecto en Xcode de forma lógica y escalable (Apple, s.f.-b).

Se recomendó utilizar nombres de archivo descriptivos, como ProductListTableViewController.swift en lugar de abreviaciones como MainVC.swift, para facilitar la navegación. Asimismo, se sugirió mantener una estructura modular, separando las definiciones de tipo en archivos independientes (Car.swift, Driver.swift, RaceTrack.swift en lugar de Model.swift combinado), lo cual favorece la reutilización y el aislamiento de responsabilidades.

Xcode permite agrupar archivos de forma visual, a través de groups, sin necesidad de reflejar esa estructura en el sistema de archivos (indicados por el icono de carpeta con sombra). Se discutió cómo usar estos grupos para mantener organizado el navegador de archivos, incluso si no coinciden exactamente con las carpetas físicas del proyecto.

Se propusieron estructuras lógicas comunes de organización como:

- ViewControllers
- Views
- Models
- Extensions
- Storyboards
- Protocols

También se enfatizó la necesidad de escribir el código pensando en la claridad y mantenimiento futuro: usando nombres explícitos, funciones pequeñas, firmas claras y comentarios útiles, pensando en que otras personas —o uno mismo en el futuro— deberán trabajar sobre esa base.

Arquitectura MVC con controladores auxiliares

Separación de responsabilidades en Model, View y Controller

El patrón Model-View-Controller (MVC) es uno de los pilares de la arquitectura de aplicaciones en UIKit. Este patrón establece una separación clara de responsabilidades entre sus componentes principales, lo que permite un diseño más mantenible, reusable y escalable (Apple, s.f.-c).

- Model: Encapsula los datos y la lógica de negocio de la aplicación. Representa el estado del sistema y puede incluir estructuras, clases o servicios de red. Es independiente de cualquier interfaz de usuario.
- View: Representa visualmente la información al usuario y capta su interacción. No contiene lógica de negocio, y su función es limitarse a la presentación.
- Controller: Actúa como intermediario entre la vista y el modelo. Controla el flujo de datos y actualiza la interfaz en respuesta a eventos del modelo o del usuario.

Esta separación permite desarrollar componentes desacoplados, facilitando las pruebas, el mantenimiento y la reutilización del código (Apple, s.f.-d). También mejora la colaboración entre desarrolladores, ya que cada elemento puede ser trabajado de forma independiente bajo responsabilidades bien definidas.

Uso de ModelControllers y HelperControllers

Para superar las limitaciones del MVC clásico, en aplicaciones complejas es común implementar extensiones de esta arquitectura como ModelControllers y HelperControllers, con el fin de mantener la separación de responsabilidades y evitar la sobrecarga del controlador de vista (Apple, s.f.-e).

- ModelControllers: Son componentes responsables de gestionar la lógica de negocio, la persistencia local y la comunicación con el backend. Actúan como puentes entre el modelo y el controlador principal.
- HelperControllers: Se utilizan para encapsular lógica secundaria o auxiliar, como validaciones, formateo de datos o servicios de red. Esto permite mantener los controladores de vista enfocados únicamente en la presentación.

El uso de estas capas intermedias fue fundamental para evitar el problema conocido como Massive View Controller y lograr una arquitectura MVC modular, mantenible y alineada con las necesidades del proyecto. Esta estructura también permitió una mayor flexibilidad para el trabajo en equipo y la implementación gradual de nuevas funcionalidades.

Comparativa con MVVM y justificación de elección

El patrón Model-View-ViewModel (MVVM) ha ganado popularidad en el desarrollo con SwiftUI debido a su capacidad para manejar el enlace de datos bidireccional y separar mejor la lógica de presentación de la vista. Sin embargo, su implementación con UIKit requiere un esfuerzo adicional en términos de infraestructura y curva de aprendizaje (Apple, s.f.-f).

Entre las ventajas de MVVM se destacan:

- Aislamiento más claro de la lógica de presentación mediante el uso de un ViewModel.
- Mejor escalabilidad en proyectos reactivos con herramientas como Combine o RxSwift.
- Reducción del acoplamiento entre vista y modelo.

No obstante, en este proyecto se optó por mantener el patrón MVC debido a varias razones técnicas y organizativas:

- UIKit fue el framework base, y su diseño está directamente alineado con MVC, facilitando la integración con herramientas nativas y simplificando la estructura general (Apple, s.f.-e).
- El equipo contaba con experiencia previa en este patrón, lo que permitió una curva de adopción más rápida sin comprometer la calidad del desarrollo.
- La incorporación de ModelControllers y HelperControllers permitió extender las ventajas de MVVM sin necesidad de abandonar MVC, conservando la claridad estructural y reduciendo la complejidad técnica del sistema.

En resumen, MVC mejorado fue la solución más adecuada al contexto, permitiendo mantener la claridad, escalabilidad y estabilidad del sistema, alineado con las capacidades del equipo y los objetivos del producto.

Patrones de comunicación entre componentes

Delegados, closures, inyección de dependencias

Durante la capacitación, se profundizó en las técnicas clásicas de comunicación entre componentes en el desarrollo con UIKit, especialmente el uso de delegados, closures e inyección de dependencias, todas ellas esenciales para mantener bajo acoplamiento y alta cohesión entre objetos en arquitectura MVC.

El patrón de delegación, ampliamente utilizado en UIKit, permite que un objeto notifique a otro sobre ciertos eventos o decisiones mediante la implementación de un protocolo. Este patrón es común en vistas como UITableView o UICollectionView, y se implementa a través de cuatro pasos: definición del protocolo, declaración de una propiedad de tipo delegado, implementación del protocolo por parte del objeto receptor, y asignación del delegado (Apple, s.f.-g). Esta técnica fue aplicada en casos prácticos de interacción entre vistas y controladores para manejar eventos de usuario, navegación y comunicación de estados internos.

```swift
protocol MyDelegate: AnyObject {
    func didTapButton()
}

class ChildViewController: UIViewController {
    weak var delegate: MyDelegate?

    @IBAction func buttonTapped(_ sender: UIButton) {
        delegate?.didTapButton()
    }
}
```

Figura 3. Implementación del patrón delegado en Swift mediante un protocolo y una clase con referencia débil.

Por su parte, los closures ofrecen una forma moderna y concisa de pasar bloques de ejecución como argumentos, facilitando el manejo de tareas asincrónicas, callbacks y actualizaciones de UI. En el contexto de MVC, los closures resultaron útiles para encapsular la lógica de presentación desacoplada del modelo. Se abordaron ejemplos donde el modelo devolvía datos a través de closures para que el controlador pudiera actualizar la interfaz sin conocer los detalles de implementación (Apple, s.f.-e).

```swift
class DataProvider {
    var onDataReady: ((String) -> Void)?

    func fetch() {
        // Simulación de obtención de datos
        onDataReady?("Datos listos")
    }
}
```

Figura 4. Uso de closures para la devolución de datos de forma asíncrona en una clase de proveedor de datos.

Finalmente, se introdujo la técnica de inyección de dependencias, orientada a mejorar la testabilidad y la modularidad del sistema. Se explicó cómo inyectar instancias directamente entre controladores para compartir información de forma explícita, evitando referencias globales o patrones singleton. Esta práctica no solo mejora la legibilidad del flujo de datos, sino que permite el uso de mocks durante pruebas unitarias o integración.

```swift
class LoginService {
    func authenticate() { /* ... */ }
}

class LoginViewController: UIViewController {
    let service: LoginService

    init(service: LoginService) {
        self.service = service
        super.init(nibName: nil, bundle: nil)
    }
}
```

Figura 5. Inyección de dependencias mediante inicializador para mejorar la modularidad y testabilidad.

Uso de Combine, tuplas y callbacks

El curso también abordó técnicas de comunicación más modernas e integradas en las arquitecturas reactivas, como el uso del framework Combine, las tuplas y los callbacks como mecanismos eficientes para transmitir datos entre componentes desacoplados.

Combine, introducido en iOS 13, permite trabajar con flujos de datos de manera declarativa y reactiva. Se presentó su uso mediante propiedades @Published en ModelControllers, a las cuales los ViewControllers se suscriben con sink para recibir actualizaciones en tiempo real. Esta estrategia facilitó la implementación de lógica reactiva sin recurrir a librerías externas como RxSwift, manteniendo la compatibilidad nativa y reduciendo la complejidad del sistema (Apple, s.f.-i).

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

let viewModel = ViewModel()
viewModel.$message.sink { print("Nuevo mensaje: \($0)") }
```

Figura 6. Observación de cambios en propiedades publicadas usando Combine y @Published.

Por otro lado, se revisó el uso de tuplas como estructuras ligeras para agrupar y transmitir múltiples valores entre funciones o componentes, sin necesidad de definir una estructura adicional.

```swift
func fetchUserData() -> (name: String, age: Int) {
    return ("Ana", 30)
}
```

Figura 7. Función que retorna una tupla nombrada con los datos de un usuario.

Los callbacks con closures también se presentaron como una alternativa común para enviar información de un controlador a otro, especialmente en flujos donde una acción debe desencadenar un resultado posterior (por ejemplo, al seleccionar un ítem y retornar la selección). Esta práctica resultó útil para evitar referencias circulares y reforzar el control del flujo de datos.

```
class DetailViewController: UIViewController {
    var onFinish: ((Bool) -> Void)?

    func done() {
        onFinish?(true)
    }
}
```

Figura 8. Uso de una closure opcional para notificar la finalización de una acción desde un controlador.

Flujo de eventos y control de dependencias

Finalmente, se abordó el diseño de un flujo de eventos desacoplado y el manejo controlado de dependencias internas. Estas dos dimensiones son esenciales para lograr un sistema modular, escalable y fácil de probar, especialmente en una arquitectura como MVC extendido con controladores auxiliares.

Durante la formación se enfatizó la importancia de centralizar la lógica de control en controladores especializados (ModelControllers), donde se produce y administra el flujo de datos, dejando a los ViewControllers únicamente la responsabilidad de presentar la información. Las herramientas revisadas, como Combine, closures y delegados, se combinaron para establecer canales de comunicación seguros y predecibles.

En cuanto a control de dependencias, se mostró cómo abstraer servicios mediante protocolos e inyectar instancias concretas en el punto de configuración de la aplicación. Este enfoque permitió reemplazar implementaciones reales por versiones simuladas o controladas durante el desarrollo y testing. En paralelo, se desaconsejó el uso de dependencias implícitas o globales que pudieran dificultar el mantenimiento y prueba del sistema.

```
protocol StorageService {
    func save(_ value: String)
}

class FileStorage: StorageService {
    func save(_ value: String) {
        print("Guardado en archivo")
    }
}
```

Figura 9. Implementación de un protocolo para definir un servicio de almacenamiento con una clase que representa un almacenamiento en archivo.

Esta visión integrada de eventos, dependencias y separación de responsabilidades sentó las bases para una arquitectura sólida y sostenible a mediano y largo plazo.

Consumo de servicios REST y asincronía

Uso avanzado de URLSession

URLSession es la clase nativa de Apple para gestionar transferencias de datos en red. Disponible desde iOS 7.0, su API ha evolucionado para incluir soporte nativo para HTTP/1.1, HTTP/2 y HTTP/3, así como tareas de descarga en segundo plano, comunicación con WebSockets y manejo seguro con políticas como ATS (App Transport Security). Es altamente configurable y adecuada para proyectos que requieren un control granular sobre las conexiones, incluyendo personalización de caché, cookies, headers, certificados y tareas en segundo plano (Apple, s.f.-j).

Se distinguieron durante la capacitación cuatro tipos principales de configuración de sesiones:

- Compartida (shared): Para solicitudes básicas sin configuración.
- Predeterminada (default): Permite delegados y configuraciones personalizadas.
- Efímera (ephemeral): No deja rastros (cookies, caché) en disco.
- Fondo (background): Permite que operaciones largas se completen aun con la app suspendida.

```
let url = URL(string: "https://api.ejemplo.com/user")!
let task = URLSession.shared.dataTask(with: url) { data, response, error in
    guard let data = data else { return }
    do {
        let user = try JSONDecoder().decode(User.self, from: data)
        print(user.name)
    } catch {
        print("Error al decodificar: \(error)")
    }
}
task.resume()
```

Figura 10. Ejemplo de solicitud HTTP con URLSession y decodificación de datos usando JSONDecoder en Swift.

```swift
func fetchUser() async {
    let url = URL(string: "https://api.ejemplo.com/user")!
    do {
        let (data, _) = try await URLSession.shared.data(from: url)
        let user = try JSONDecoder().decode(User.self, from: data)
        print(user.name)
    } catch {
        print("Error: \(error)")
    }
}
```

Figura 11. Solicitud de red moderna usando async/await y decodificación con JSONDecoder en Swift.

Asimismo, se revisaron las principales tareas de red que URLSession puede ejecutar:

- DataTask: Para envíos y respuestas simples.
- UploadTask: Para subir archivos grandes o en segundo plano.
- DownloadTask: Para gestionar grandes descargas con seguimiento.
- WebSocketTask: Para comunicación bidireccional en tiempo real.

El uso de delegados para URLSession permite interceptar eventos como autenticación, recepción de datos incremental, manejo de caché y reintentos, sin bloquear la interfaz. Esto es especialmente útil en flujos donde se requiere responder a eventos asincrónicos desde múltiples capas del sistema.

```swift
class NetworkDelegate: NSObject, URLSessionDelegate {
    func urlSession(_ session: URLSession, didBecomeInvalidWithError error: Error?) {
        print("Sesión inválida: \(error?.localizedDescription ?? "Sin error")")
    }
}

let session = URLSession(configuration: .default, delegate: NetworkDelegate(), delegateQueue: nil)
```

Figura 12. Uso de un delegado personalizado (URLSessionDelegate) para manejar eventos de invalidación en sesiones de red.

Comparativa entre URLSession y Alamofire

Durante la formación, también se analizó la biblioteca de terceros Alamofire, ampliamente utilizada en proyectos iOS para simplificar la interacción con APIs RESTful. Alamofire está construido sobre URLSession, pero ofrece una interfaz más declarativa, funcionalidades integradas de serialización, interceptores, validación de certificados y control de errores más compacto (Apple, s.f.-j).

Criterio	URLSession	Alamofire
Naturaleza	API nativa de Apple	Biblioteca externa construida sobre URLSession
Flexibilidad	Máximo control, requiere más código	Simplicidad, menor control de bajo nivel
Serialización	Manual (usualmente con JSONDecoder)	Automática con métodos integrados (responseDecodable)
Interceptores	Requiere configuración propia	Integrados para headers, autenticación, logging
Casos de uso ideal	Proyectos que requieren control detallado	Proyectos que privilegian velocidad y simplicidad de desarrollo
Comunidad y soporte	Oficial de Apple	Comunidad activa de código abierto

Tabla 1. Comparativa entre URLSession y Alamofire en proyectos iOS.

Ambas opciones son válidas y potentes. La elección depende del contexto: si se necesita una solución liviana, segura, nativa y sin dependencias externas, URLSession es preferible. En cambio, si se prioriza la velocidad de desarrollo, legibilidad del código y modularidad, Alamofire ofrece ventajas inmediatas.

```
AF.request("https://api.ejemplo.com/user")
    .validate()
    .responseDecodable(of: User.self) { response in
        switch response.result {
        case .success(let user):
            print(user.name)
        case .failure(let error):
            print("Error: \(error)")
        }
    }
```

Figura 13. Ejemplo de uso de Alamofire para realizar una petición HTTP y decodificar automáticamente la respuesta utilizando responseDecodable.

Programación con GCD y async/await

Para aprovechar el comportamiento asíncrono de URLSession y otras operaciones intensivas, se capacitaron dos enfoques esenciales: Grand Central Dispatch (GCD) y el nuevo modelo de concurrencia basado en async/await, introducido en Swift 5.5.

GCD fue presentado como una solución madura para gestionar tareas concurrentes mediante colas de ejecución (DispatchQueue). Se diferenciaron las colas seriales (una tarea a la vez) y colas concurrentes

(varias tareas en paralelo), y se destacó la importancia de enviar actualizaciones de UI al hilo principal (DispatchQueue.main.async) tras completar operaciones en segundo plano.

```
DispatchQueue.global(qos: .background).async {
    let result = longRunningTask()
    DispatchQueue.main.async {
        self.label.text = result
    }
}
```

Figura 14. Ejecución de una tarea en segundo plano con DispatchQueue y actualización posterior en el hilo principal.

Async/await, por su parte, ofrece una sintaxis declarativa y lineal para manejar código asincrónico, eliminando el uso excesivo de closures o callbacks anidados. En combinación con URLSession.shared.data(for:delegate:), es posible realizar peticiones de red que se leen y se controlan como si fueran secuenciales. Este modelo mejora la legibilidad, permite una mejor propagación de errores mediante do/try/catch, y se considera una práctica recomendada para proyectos modernos (Apple, 2021c; Apple, 2021d).

```
func loadData() async {
    do {
        let (data, _) = try await URLSession.shared.data(from: URL(string: "https://api.ejemplo.com/data")!)
        let result = try JSONDecoder().decode(MyModel.self, from: data)
        print(result)
    } catch {
        print("Falló la solicitud: \(error)")
    }
}
```

Figura 15. Petición asincrónica utilizando async/await con URLSession y decodificación de respuesta usando JSONDecoder.

A modo de conclusión, se recomendó el uso de async/await como estándar para proyectos modernos en iOS 15+, manteniendo GCD como una herramienta útil para flujos más bajos, como sincronización de procesos o ejecución paralela.

Gestión segura y eficiente del almacenamiento local

UserDefaults, Keychain, FileManager y Core Data

Apple ofrece una serie de tecnologías nativas para la persistencia local de datos en aplicaciones desarrolladas con UIKit. Cada herramienta tiene ventajas específicas según el tipo de información que se desea almacenar, la frecuencia de acceso, los requisitos de seguridad y la necesidad de sincronización entre dispositivos.

UserDefaults: La clase UserDefaults permite almacenar de forma sencilla datos como preferencias de usuario, configuraciones o banderas booleanas. Es ideal para persistencia ligera y se sincroniza automáticamente entre ejecuciones de la aplicación. Está disponible en todas las plataformas del ecosistema Apple. Sin embargo, su uso no es recomendable para datos sensibles ni para estructuras complejas o grandes volúmenes de información (Apple, s.f.-k).

Keychain: Para información sensible, como contraseñas o tokens de sesión, la herramienta más segura es el Keychain. Su cifrado está integrado en el sistema y los datos persisten incluso después de que la aplicación sea desinstalada. Aunque su API es más compleja y de bajo nivel, permite compartir datos entre apps del mismo desarrollador cuando se configura adecuadamente (Apple, s.f.-l).

FileManager: FileManager proporciona una interfaz para leer y escribir directamente en el sistema de archivos del dispositivo. Es útil para el manejo de documentos, imágenes, vídeos u otros archivos generados por el usuario o descargados. Permite definir estructuras de carpetas personalizadas, pero requiere una gestión manual y cuidadosa de rutas, permisos y buenas prácticas, como evitar almacenar datos temporales en el directorio Documents (Apple, s.f.-m).

Core Data: Core Data es el framework de persistencia más robusto del ecosistema Apple. Se trata de un ORM (Object Relational Mapping) que permite gestionar modelos de datos complejos y relaciones entre entidades. Incluye herramientas para migración de esquemas, versionado y sincronización mediante iCloud. Su curva de aprendizaje es más pronunciada, pero es ideal para aplicaciones que manejan grandes volúmenes de información o estructuras relacionales sofisticadas (Apple, s.f.-n).

Estrategias para persistencia segura y sincronización

Durante el desarrollo profesional y la capacitación técnica, se resaltó la importancia de seleccionar el mecanismo de almacenamiento apropiado para cada tipo de dato. En general, se establecieron las siguientes directrices:

- Usar UserDefaults para almacenar configuraciones simples no sensibles.
- Emplear Keychain para datos que involucren seguridad, autenticación o identidad.
- Utilizar FileManager para archivos grandes o estructuras personalizadas de documentos del usuario.
- Implementar Core Data o, en su caso, SQLite, para modelos complejos con relaciones y operaciones frecuentes.
- Integrar iCloud Documents cuando sea necesario mantener sincronización entre dispositivos Apple sin intervención manual.

Estas estrategias aseguran un equilibrio entre seguridad, eficiencia, escalabilidad y compatibilidad multiplataforma. A lo largo del proceso de desarrollo, se hizo énfasis en seguir las guías de

almacenamiento seguro de Apple y en diseñar flujos que prevengan pérdidas de información, corrupción de archivos o duplicidad de datos.

Diseño de interfaces en UIKit

Principios de diseño con Storyboards y Auto Layout

UIKit continúa siendo el framework más maduro y ampliamente adoptado para el diseño de interfaces gráficas en aplicaciones iOS, iPadOS y macOS. A diferencia de SwiftUI, su paradigma imperativo proporciona control detallado sobre el ciclo de vida de los controladores de vista y su integración con UIViewController, UINavigationController y UITabBarController.

Durante la capacitación se abordaron principios fundamentales del diseño en UIKit, haciendo énfasis en el uso de Storyboards y Auto Layout. Storyboards permiten construir visualmente la estructura de navegación y las relaciones entre pantallas, mientras que Auto Layout facilita la creación de interfaces responsivas, adaptables a distintas resoluciones y dispositivos, respetando márgenes seguros y comportamientos dinámicos (Apple, s.f.-e).

Este enfoque visual no solo mejora la legibilidad del flujo de pantallas, sino que permite reutilizar componentes y establecer constraints precisas sin tener que escribir código adicional, algo especialmente útil en proyectos con múltiples desarrolladores o iteraciones rápidas.

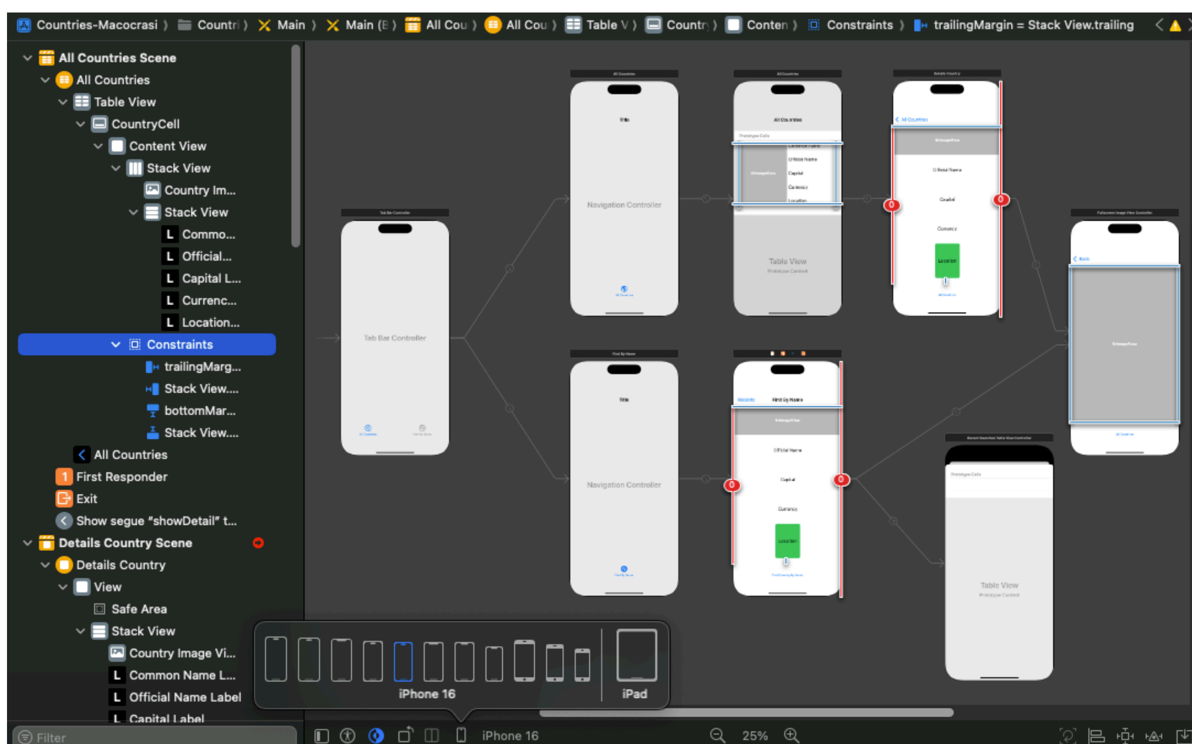


Figura 16. Ejemplo Storyboard con Auto Layout y distintos dispositivos.

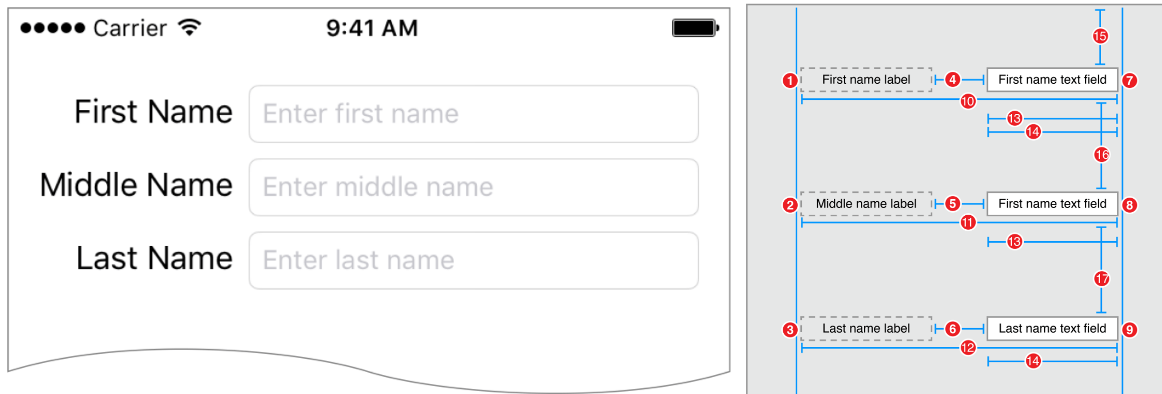


Figura 17. Ejemplo de vista con contenido intrínseco usando Auto Layout - Fuente: Apple (s.f.-o)

Nota: Para la Figura 17, las siguientes reglas de Auto Layout fueron aplicadas a los elementos visuales enumerados en la imagen de la derecha:

1. First Name Label.Leading = Superview.LeadingMargin
2. Middle Name Label.Leading = Superview.LeadingMargin
3. Last Name Label.Leading = Superview.LeadingMargin
4. First Name Text Field.Leading = First Name Label.Trailing + Standard
5. Middle Name Text Field.Leading = Middle Name Label.Trailing + Standard
6. Last Name Text Field.Leading = Last Name Label.Trailing + Standard
7. First Name Text Field.Trailing = Superview.TrailingMargin
8. Middle Name Text Field.Trailing = Superview.TrailingMargin
9. Last Name Text Field.Trailing = Superview.TrailingMargin
10. First Name Label.Baseline = First Name Text Field.Baseline
11. Middle Name Label.Baseline = Middle Name Text Field.Baseline
12. Last Name Label.Baseline = Last Name Text Field.Baseline
13. First Name Text Field.Width = Middle Name Text Field.Width
14. First Name Text Field.Width = Last Name Text Field.Width
15. First Name Text Field.Top = Top Layout Guide.Bottom + 20.0
16. Middle Name Text Field.Top = First Name Text Field.Bottom + Standard
17. Last Name Text Field.Top = Middle Name Text Field.Bottom + Standard

Patrones de navegación en UIKit

UIKit proporciona múltiples mecanismos de navegación entre controladores de vista que permiten construir flujos de trabajo coherentes, escalables y adaptables a distintas plataformas del ecosistema Apple. Durante la capacitación, se abordaron tres pilares fundamentales: navegación programática, navegación mediante Storyboards con segues, y navegación estructurada por secciones utilizando UITabBarController.

Uno de los patrones más comunes en aplicaciones iOS es la navegación en pila (push), gestionada a través de un UINavigationController. Este controlador permite apilar vistas de forma jerárquica y presentar nuevos controladores mediante el método pushViewController(_:animated:). Se explicó también el uso de transiciones modales para presentar vistas fuera del flujo jerárquico, así como la combinación de ambas estrategias para manejar flujos dependientes y flujos paralelos.

En el contexto visual, los segues definen conexiones entre pantallas dentro de Storyboards. Se revisaron en detalle los métodos prepare(for:sender:), performSegue(withIdentifier:sender:) y shouldPerformSegue(withIdentifier:sender:), esenciales para pasar datos entre vistas, condicionar transiciones y asegurar la integridad del flujo.

Complementariamente, se profundizó en el uso de la barra de navegación (UINavigationBar), sus elementos (navigationItem, UIBarButtonItem), y los patrones de apariencia global mediante UINavigationBar.appearance() para mantener consistencia visual. Esta personalización es especialmente importante en aplicaciones que demandan una identidad gráfica propia o integración con flujos administrativos y herramientas empresariales.

Otro componente clave revisado fue el UITabBarController, utilizado para organizar múltiples secciones de la aplicación en una estructura horizontal. Su implementación fue abordada tanto desde el Storyboard como desde código, utilizando arreglos de controladores e íconos personalizados (UITabBarItem). Esta interfaz es común en aplicaciones que agrupan contenido por dominios funcionales: inicio, servicios, configuración, etc.

Asimismo, se revisó el protocolo UITabBarControllerDelegate para gestionar cambios de pestaña y detectar el comportamiento activo del usuario. Este nivel de control permite ejecutar operaciones asincrónicas o cambiar el estado de la aplicación de forma dinámica al detectar la pestaña seleccionada (Apple, s.f.-f).

Finalmente, se exploró la implementación de transiciones unwind, las cuales permiten regresar desde una vista secundaria a una anterior de manera controlada. Estas transiciones son útiles cuando no se desea desapilar vistas manualmente ni depender exclusivamente de botones de navegación.

Al integrar storyboards visuales, navegación programática y componentes estándar como barras y tab bars, se logra una experiencia de usuario clara y predecible. Estos patrones permiten separar la lógica de flujo, mantener estructuras mantenibles, y asegurar que las aplicaciones respondan tanto a criterios de usabilidad como a necesidades empresariales.

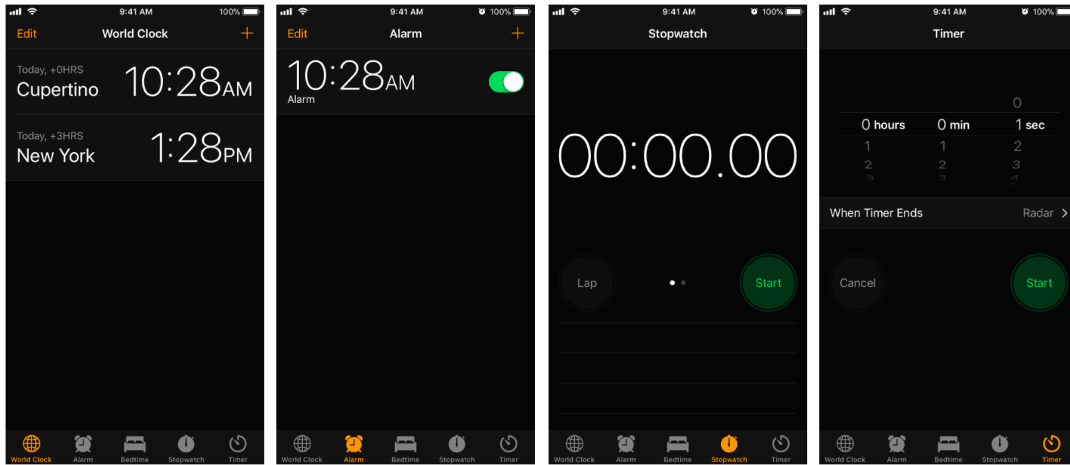


Figura 18. Ejemplo del uso de UITabBarController con múltiples secciones en una aplicación de reloj. Fuente: Apple (s.f.-p)

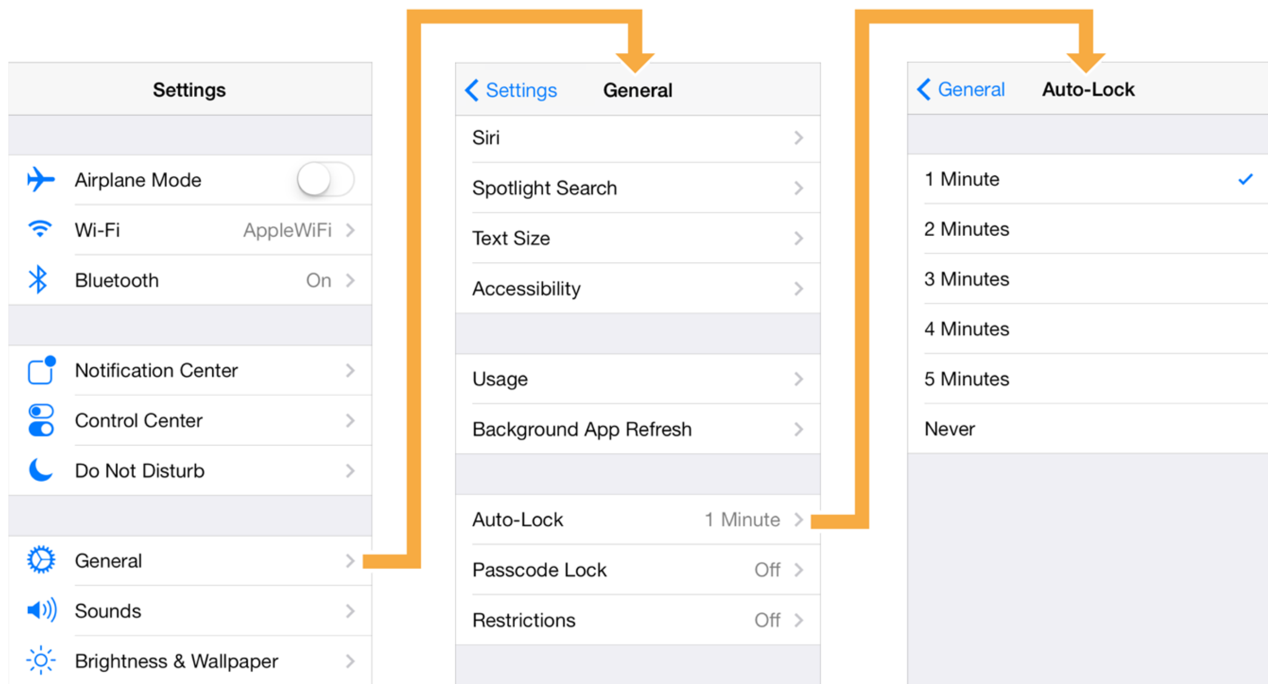


Figura 19. Ejemplo de navegación jerárquica con UINavigationController en la app de Configuración. Fuente: Apple (s.f.-q)

Control de versiones y flujo de trabajo con GitFlow

Tipos de ramas: main, develop, feature, release y hotfix

Git Flow es una estrategia de control de versiones que organiza el desarrollo de software mediante una estructura clara y jerárquica de ramas. Fue introducida en 2010 por Vincent Driessen y ha sido ampliamente adoptada por equipos de desarrollo que requieren flujos controlados para entornos colaborativos, con múltiples fases de desarrollo, pruebas y lanzamiento.

El flujo se basa en cinco tipos de ramas:

- **main**: Esta rama contiene siempre el código estable que está listo para producción. Cada commit en main debe representar una versión que pueda ser desplegada en cualquier momento. Por convención, se mantiene libre de errores y solo recibe código probado y validado.
- **develop**: Es la rama base para el desarrollo de nuevas funcionalidades. Aquí se integran las ramas de feature y representa el estado pre-producción de la aplicación. Aunque es menos estable que main, debe mantener un nivel funcional suficiente para pruebas de integración.
- **feature**: Se crean ramas de feature a partir de develop para trabajar en nuevas funcionalidades de forma aislada. Una vez implementada y revisada, la rama se fusiona nuevamente en develop. Esta separación evita contaminar el código base con cambios incompletos o experimentales.
- **release**: Estas ramas se derivan de develop cuando se desea preparar una nueva versión estable. En ellas se realizan ajustes menores, corrección de errores y documentación final antes del despliegue. Al concluir, la rama se fusiona tanto en main como en develop.
- **hotfix**: Son ramas temporales que se crean directamente desde main para resolver errores críticos en producción. Posteriormente se integran en main y develop para mantener la coherencia de la base de código en todas las ramas activas.

Esta estructura permite un control riguroso sobre qué tipo de código se encuentra en cada fase del ciclo de desarrollo, minimizando errores en producción y facilitando la gestión de versiones.

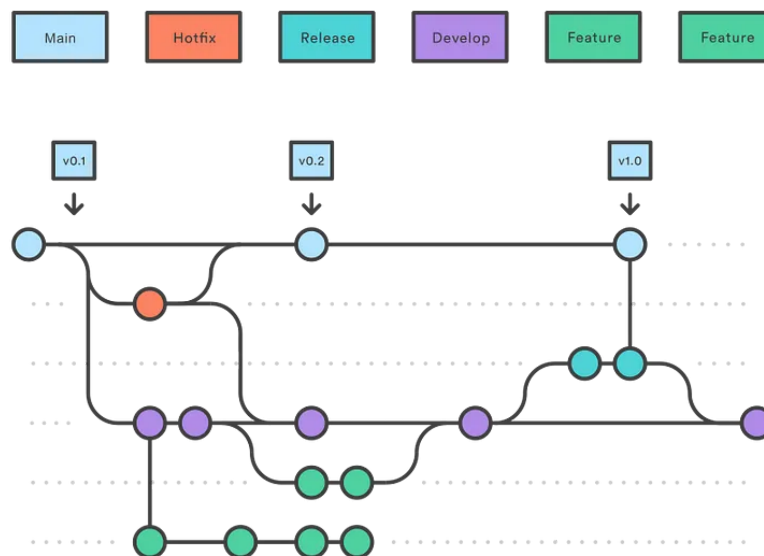


Figura 20. Diagrama GitFlow

Nota: La Figura 20 ilustra el modelo GitFlow aplicado, donde cada tipo de rama cumple una función específica dentro del flujo de trabajo. Se observa cómo las ramas feature se originan desde develop, y posteriormente se integran para formar versiones candidatas en release, antes de pasar a main. Las ramas hotfix se desprenden directamente de main para corregir errores críticos y se reintegran tanto a main como a develop, asegurando la consistencia del historial. Las etiquetas v0.1, v0.2 y v1.0 muestran ejemplos de versionado semántico aplicado durante el ciclo de vida del desarrollo.
Fuente: Sergio Humberto. (s.f.)

Buenas prácticas para mantenimiento, integración y revisión

Implementar Git Flow eficazmente requiere adoptar una serie de buenas prácticas que aseguren la calidad, estabilidad y trazabilidad del código.

- La rama main debe mantenerse limpia y únicamente contener código validado y probado. Se recomienda que las fusiones hacia esta rama estén sujetas a revisiones estrictas y pruebas automatizadas.
- La rama develop debe ser la única base para nuevas funcionalidades. Esto garantiza una separación clara entre el código en desarrollo y el código estable. Cada feature debe desarrollarse en una rama independiente, preferentemente con nombres descriptivos y relacionados al objetivo de la tarea.
- Las ramas de feature y release deben ser eliminadas una vez fusionadas para evitar confusión y mantener el repositorio limpio. Asimismo, se aconseja que las ramas de feature nunca se fusionen directamente en main.
- En el caso de hotfix, su uso debe reservarse exclusivamente para incidentes urgentes que requieran atención inmediata en producción. Su correcta integración en develop evita que se pierdan correcciones en futuras versiones.
- Se recomienda llevar un proceso de revisión de código formal antes de cualquier fusión importante. Esta revisión puede realizarse mediante pull requests, y debe incluir comentarios técnicos, validaciones de estilo y pruebas de funcionamiento.
- Una comunicación efectiva dentro del equipo es esencial. Todos los miembros deben estar al tanto del estado de las ramas, cambios estructurales y convenciones adoptadas.
- Finalmente, es fundamental documentar tanto en los mensajes de commit como en el repositorio principal cualquier decisión técnica o cambio relevante en el flujo de trabajo. Esto facilita el seguimiento histórico del proyecto.

Uso de etiquetas, CI/CD y pruebas automatizadas con XCTest

Para complementar el flujo de trabajo de Git Flow y asegurar la calidad del software en cada etapa, se recomienda integrar herramientas de etiquetado, pruebas automatizadas y despliegue continuo.

- Las etiquetas (tags) son utilizadas para marcar versiones específicas en ramas de release y main. Esto permite identificar y rastrear versiones distribuidas, generar changelogs, o realizar rollbacks en caso de ser necesario.
- En paralelo, se fomenta la implementación de procesos de Integración Continua / Despliegue Continuo (CI/CD). Este enfoque permite que cada commit en ramas críticas dispare automáticamente procesos de construcción, validación y despliegue en entornos de pruebas o producción.
- En el contexto de desarrollo iOS, se utilizaron pruebas unitarias mediante el framework XCTest, integradas dentro del pipeline de CI. Estas pruebas permiten detectar errores tempranamente, asegurar el funcionamiento de nuevos módulos, y evitar regresiones en funcionalidades existentes.
- Se estableció un sistema donde cada pull request debía ejecutar satisfactoriamente una batería de pruebas automatizadas antes de ser integrado en develop o main. Esto ayudó a mantener la integridad del proyecto, incluso con múltiples desarrolladores trabajando en paralelo.

La combinación de etiquetado semántico, pruebas automatizadas y validación continua refuerza el objetivo principal de Git Flow: garantizar que el código en producción sea siempre estable, reproducible y confiable.

Capítulo 3 - Implementación práctica y validación funcional

Despliegue inicial del proyecto y aplicación de arquitectura modular

La puesta en marcha del proyecto Wee 3.0 comenzó con el establecimiento de un repositorio centralizado en Azure DevOps, utilizando una metodología GitFlow para gestionar las ramas, versionar los cambios y facilitar las revisiones colaborativas dentro del equipo. Esta configuración se complementó con el uso de GitKraken, una herramienta gráfica que permitió organizar el flujo de trabajo de manera visual y estructurada, integrando prácticas como feature branching, pull requests y release tracking.

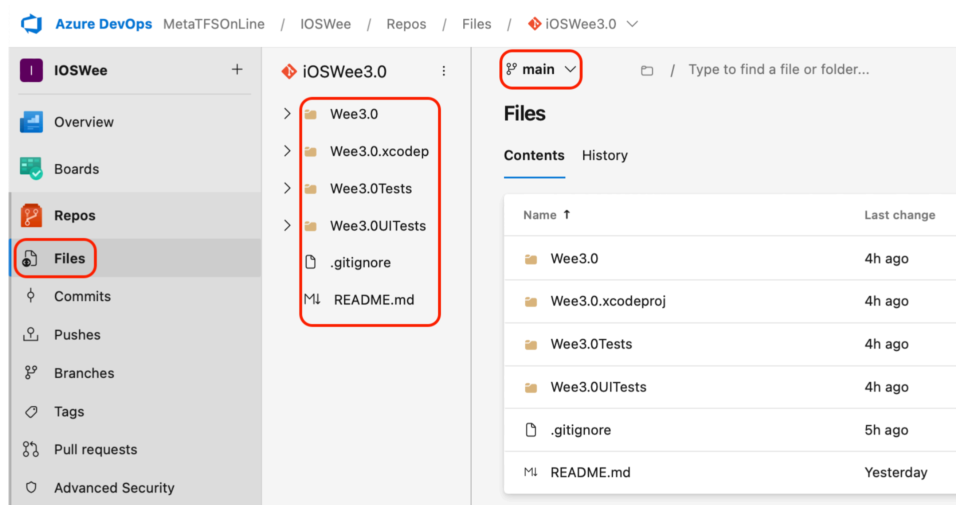


Figura 21. Estructura inicial del repositorio del proyecto Wee 3.0 en Azure DevOps, rama principal main.

Una vez creado el repositorio, se inició la conexión entre Azure DevOps y GitKraken, generando las credenciales necesarias para la autenticación mediante tokens de acceso personal. Se configuraron las ramas principales (main y develop) y se habilitaron las políticas de protección de ramas, incluyendo:

- Revisión obligatoria por un miembro del equipo antes de fusionar.
- Asociación de cambios con work items (historias de usuario).
- Resolución documentada de comentarios antes del merge.

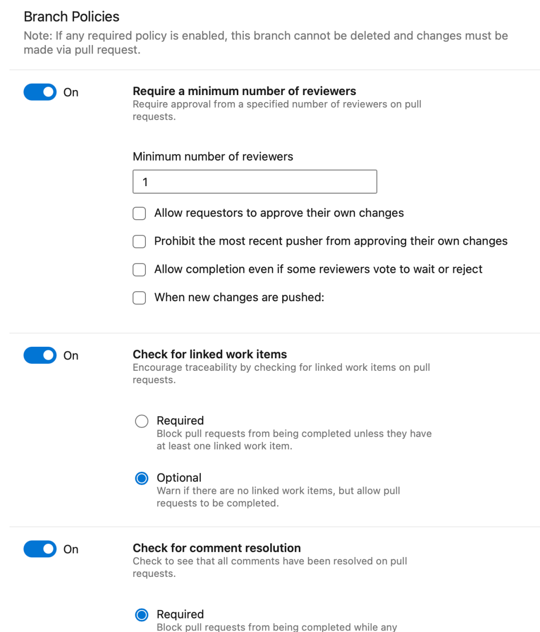


Figura 22. Políticas de protección configuradas para la rama main en Azure DevOps: revisión obligatoria, trazabilidad de historias y resolución de comentarios antes del merge.

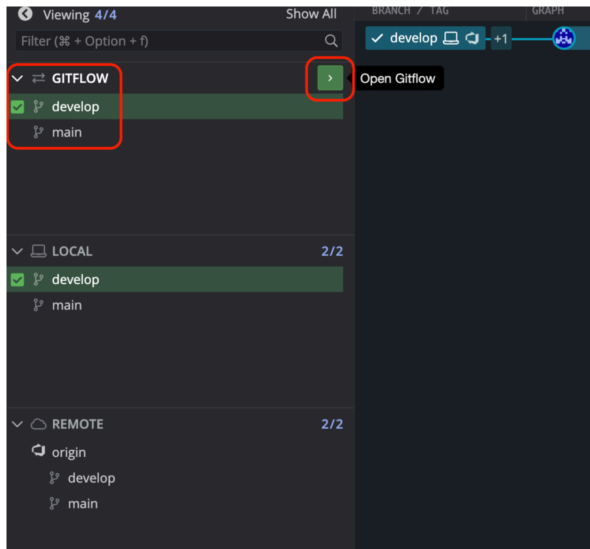


Figura 23. Vista de ramas main y develop gestionadas mediante GitFlow en la interfaz de GitKraken.

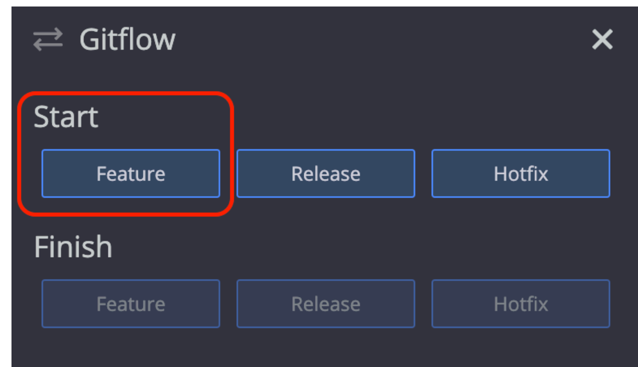


Figura 24. Menú de creación de ramas de tipo feature, release y hotfix en GitKraken mediante flujo GitFlow.

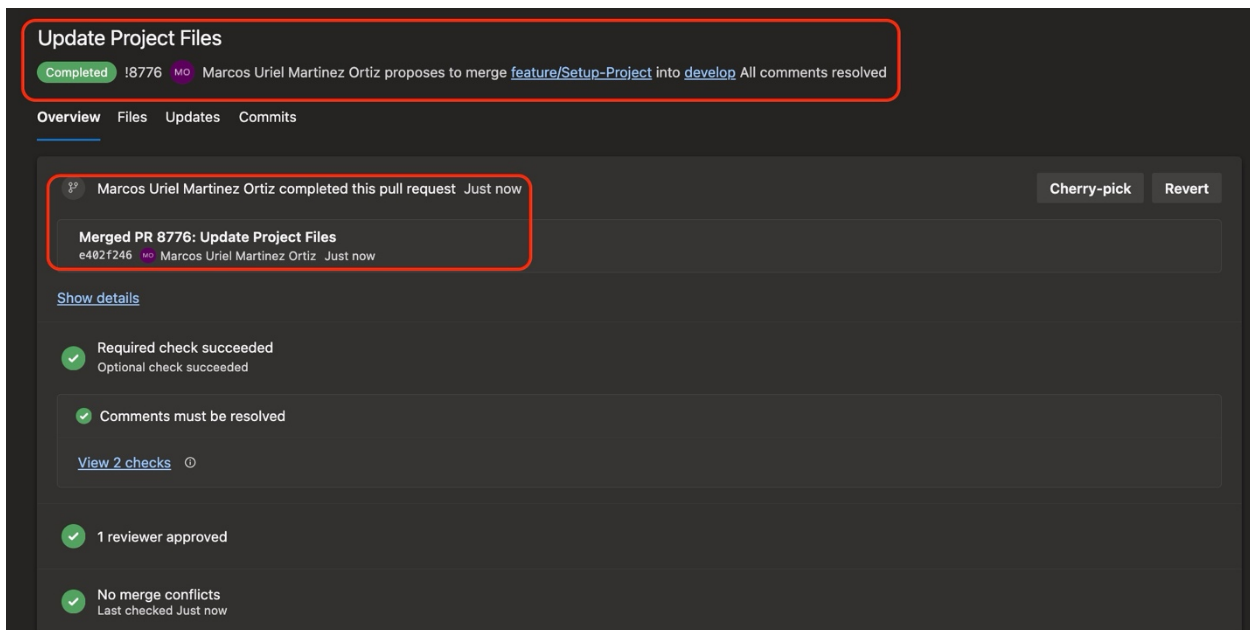


Figura 25. Pull Request completado exitosamente en Azure DevOps, mostrando la integración de la rama feature/Setup-Project a develop con validaciones aprobadas y sin conflictos.

Con estas políticas activas, se creó el proyecto Wee 3.0 en Xcode, utilizando Storyboard como sistema de interfaz gráfica, e incluyendo archivos de pruebas automatizadas (XCTest) desde la fase inicial. El proyecto fue estructurado con una jerarquía clara de carpetas, siguiendo un enfoque modular:

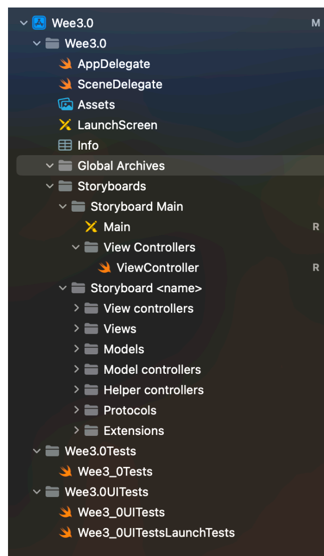


Figura 26. Estructura modular del proyecto Wee 3.0 en Xcode, organizada por Storyboards, controladores, modelos y pruebas.

Además, se incorporó una carpeta de uso global denominada Global Archives, la cual contenía plantillas y componentes reutilizables compartidos entre módulos.

La carga inicial del proyecto se realizó siguiendo los principios de GitFlow: creación de rama feature, subida de archivos mediante push local, y solicitud de pull request revisada y aprobada por otro integrante del equipo. Posteriormente, se realizó un merge hacia develop, y una vez consolidada la arquitectura base, se creó una rama release que fue integrada finalmente en main, completando el despliegue oficial del proyecto.

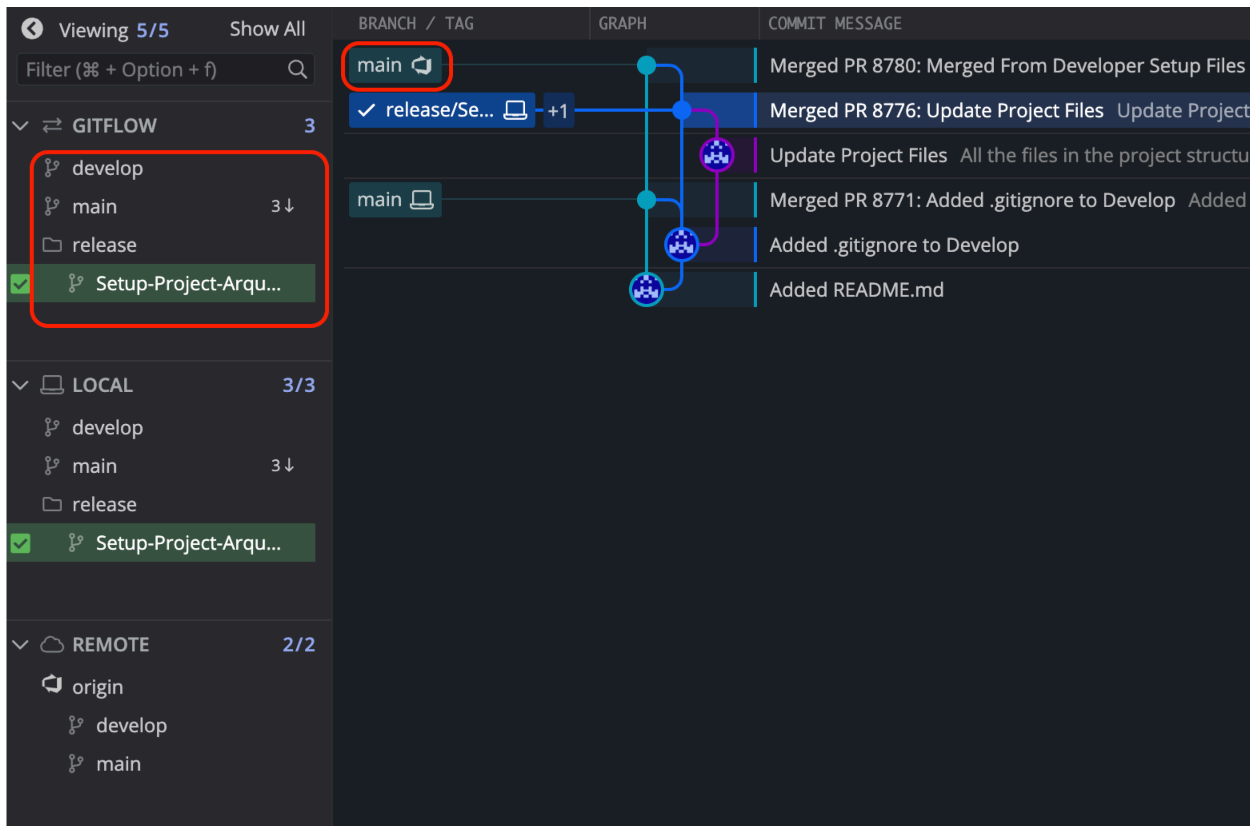


Figura 27. Flujo de ramas y confirmaciones en GitKraken durante la fase inicial del proyecto Wee 3.0: integración de la arquitectura base desde Setup-Project-Arquitecture hacia develop, release y main.

Este proceso, además de garantizar la trazabilidad y validación del código, sentó las bases para un flujo de trabajo sostenible, con ciclos de integración continua, mantenimiento simplificado y colaboración efectiva en equipo.

Punto de entrada y arquitectura de entorno en el inicio de sesión

El nuevo inicio de sesión se estableció como el punto de entrada fundamental de la aplicación dentro del Storyboard Main. A diferencia de versiones anteriores, esta pantalla no solo marca el acceso del usuario, sino que representa un cambio estructural clave en la arquitectura general del proyecto Wee 3.0. Desde su

implementación, se incorporó un sistema de archivos de configuración segmentados por entorno, con esquemas independientes para desarrollo y producción. Esto permitió centralizar variables críticas como URLs del backend, claves de API, y credenciales de autenticación, facilitando el cambio de ambiente sin modificar el código fuente.

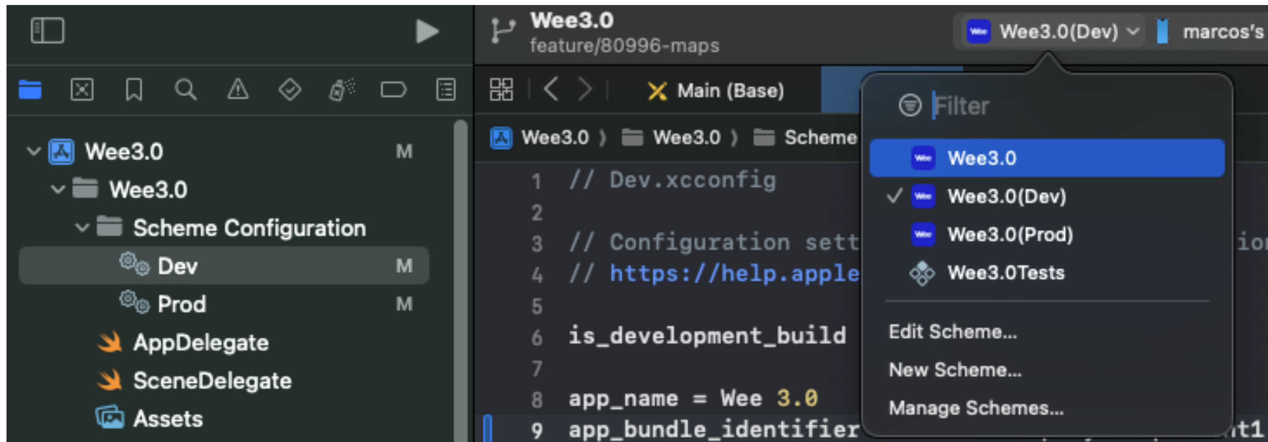


Figura 28. Configuración de esquemas de entorno (Dev y Prod) en Xcode para la aplicación Wee 3.0.

Nota: Esta configuración permite alternar fácilmente entre ambientes de desarrollo y producción desde el entorno de ejecución en Xcode, utilizando archivos .xcconfig para definir variables globales como el identificador del bundle, nombre de la app y URLs del backend.

Gracias a esta arquitectura, los procesos de prueba por parte de la Product Owner (PO), quien también desempeñaba funciones de QA, se simplificaron significativamente. Ya no era necesario generar múltiples versiones de la aplicación para cada entorno: bastaba con ejecutar el proyecto bajo el esquema deseado para realizar pruebas completas y consistentes desde el flujo de inicio de sesión.

Figura 29. Pantalla de inicio de sesión con selector oculto de entorno en Wee 3.0.

Nota: Al realizar una presión prolongada sobre el botón “Iniciar sesión”, se despliega un menú oculto que permite seleccionar el entorno (Desarrollo, QA, Producción, etc.). Esta funcionalidad está restringida mediante archivos de configuración (.xcconfig) para que no se active en la versión de producción, y fue diseñada para facilitar las pruebas manuales sin necesidad de múltiples versiones de la app.

Adicionalmente, el diseño de esta pantalla —y del proyecto en general— se concibió desde el inicio como multidispositivo y adaptable, compatible con todos los tamaños y orientaciones de pantalla en iOS, iPadOS y macOS mediante Catalyst. El uso de Auto Layout y un sistema visual desacoplado a través de Storyboard References garantizó que esta primera vista pudiera escalar correctamente, sin distorsiones, en cualquier dispositivo del ecosistema Apple.

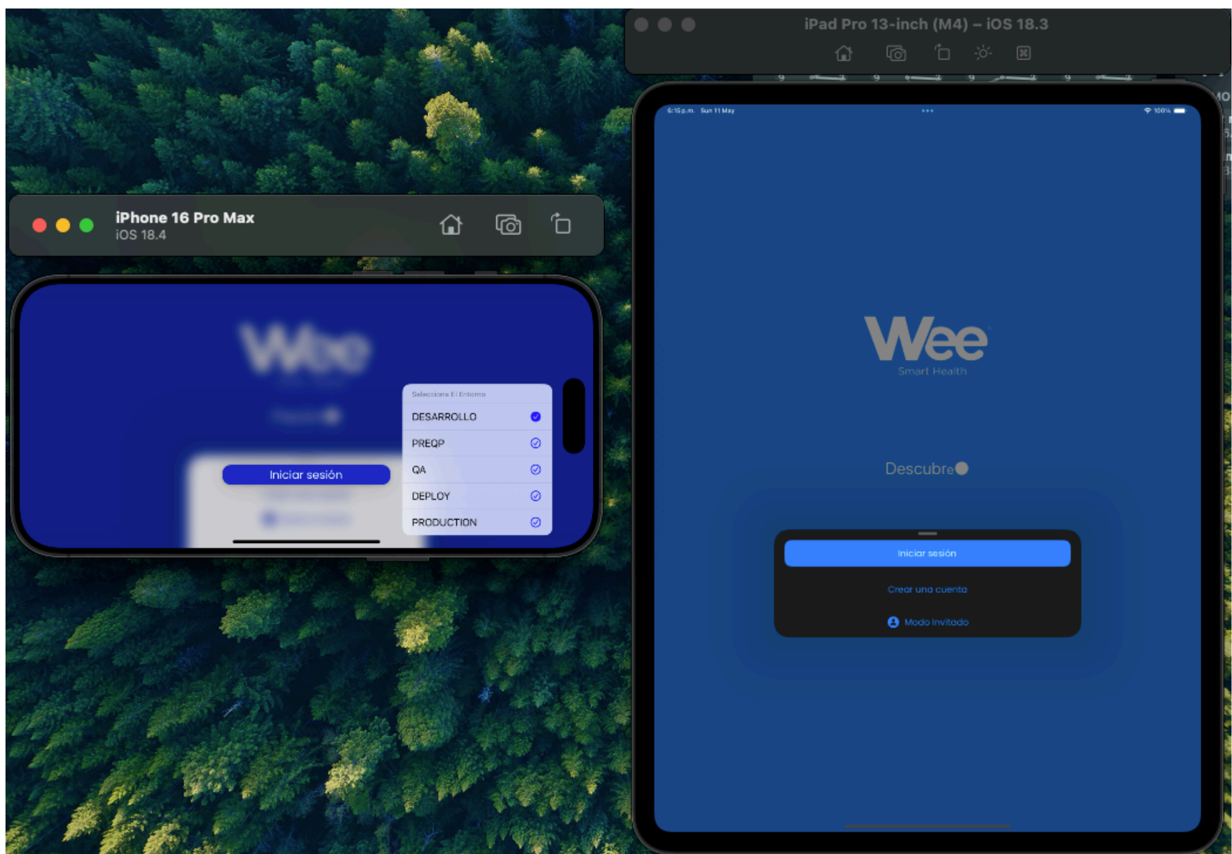
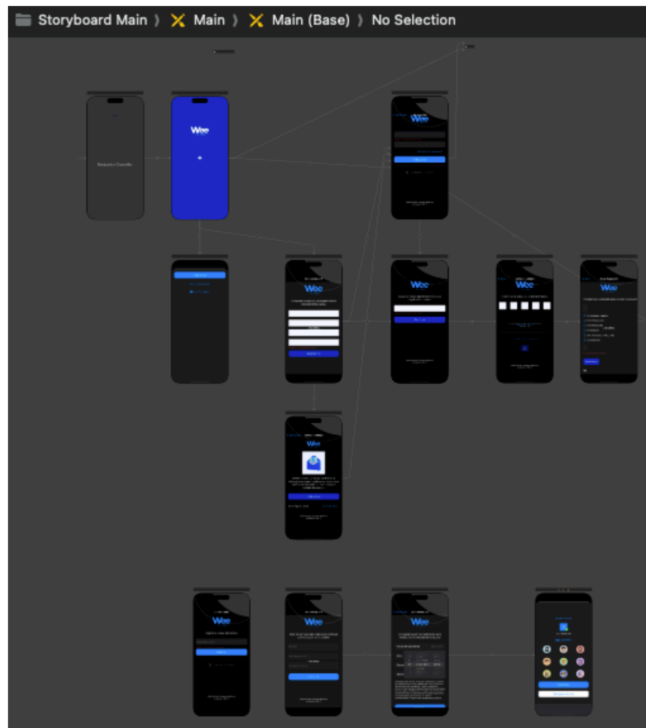


Figura 30. Compatibilidad visual del login en iPhone y iPad usando un único storyboard adaptado.

Nota: La misma pantalla de inicio de sesión se muestra correctamente tanto en un iPhone 16 Pro Max en orientación horizontal como en un iPad Pro de 13 pulgadas con modo oscuro activado. Esta adaptabilidad se logró exclusivamente mediante Auto Layout y el uso de componentes nativos, sin necesidad de duplicar vistas ni condicionar el flujo para cada plataforma.



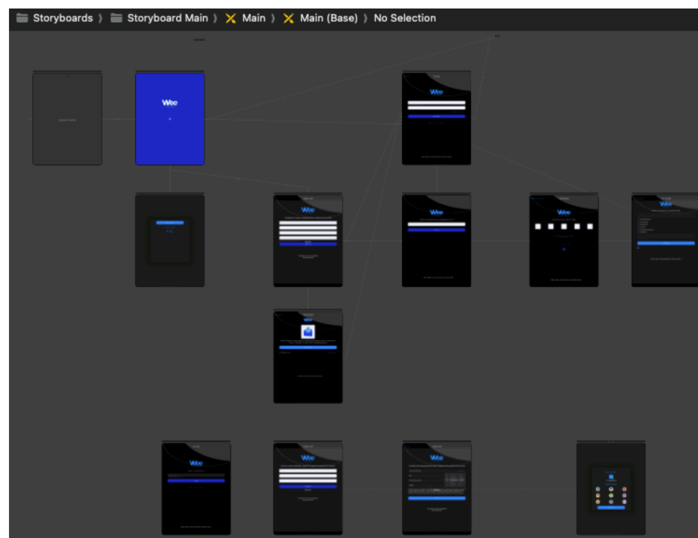


Figura 31. Visualización del flujo de inicio de sesión desde Storyboard en iPhone 16 Pro Max y iPad Pro 13".

Nota: La parte superior muestra el flujo renderizado en un iPhone 16 Pro Max, mientras que la parte inferior corresponde a un iPad Pro de 13" (M4). Esta visualización, tomada directamente desde el Storyboard de Xcode, permite verificar en tiempo real la correcta adaptación de los elementos a distintas resoluciones y orientaciones, lo cual agiliza las pruebas de interfaz sin necesidad de ejecutar la app.

Modularización de interfaces con Storyboard References

En el desarrollo de Wee 3.0, uno de los mayores desafíos fue mantener la modularidad visual, el orden jerárquico de navegación y la colaboración fluida entre desarrolladores sin perder los beneficios del diseño visual proporcionado por Interface Builder. Para resolver esta problemática de forma elegante y nativa, se optó por un enfoque basado en Storyboard References, una funcionalidad introducida en iOS 9 que permite dividir un storyboard principal en varios storyboards secundarios, sin renunciar a la capacidad de enlazar escenas visualmente mediante segues.

¿Qué es una Storyboard Reference?

Una Storyboard Reference funciona como un "puente visual" entre archivos storyboard distintos. Actúa como un placeholder dentro del storyboard principal, apuntando a un ViewController que vive en otro archivo storyboard del mismo proyecto. Esta estructura permite que Xcode compile y valide los enlaces como si formaran parte del mismo documento, pero internamente están separados a nivel de archivo. Esto habilita una arquitectura limpia, visual y escalable sin necesidad de transiciones programáticas entre controladores.

Ventajas de usar Storyboard References

1. **Modularidad y escalabilidad:** Dividir la interfaz en múltiples storyboards facilita el mantenimiento y la navegación entre flujos complejos. Por ejemplo, en Wee 3.0 se aislaron flujos como autenticación, recuperación de contraseña y panel de usuario.

2. **Colaboración sin conflictos:** En lugar de que dos desarrolladores editen el mismo storyboard (archivo XML), cada quien puede trabajar sobre su storyboard independiente, evitando conflictos en Git, uno de los mayores problemas de los storyboards monolíticos.

3. **Reutilización y testeo independiente:** Al separar vistas o flujos reutilizables (por ejemplo, un formulario de contacto o un verificador de sesión), estos se pueden mantener, probar y versionar de forma independiente.

4. **Navegación fluida y validación de vínculos:** Los segues entre storyboards son totalmente compatibles con Interface Builder, y si hay errores en los IDs o enlaces mal configurados, Xcode los detectará en tiempo de compilación, evitando errores en tiempo de ejecución.

Ventajas frente al diseño 100% programático

Aunque crear interfaces mediante código (con NSLayoutConstraint, UIView(), etc.) permite control absoluto sobre la lógica visual, este enfoque tiene costos importantes en tiempo, legibilidad y mantenibilidad. En contraste, el uso de Storyboards, reforzado con Storyboard References, ofrece:

- Visualización inmediata de flujos completos sin necesidad de compilar.
- Compatibilidad con múltiples dispositivos y orientaciones gracias a Auto Layout y Size Classes.
- Reutilización de interfaces sin duplicar código.
- Transiciones visuales configurables, sin necesidad de lógica adicional.

Esta estrategia también permitió que diseñadores visuales o ingenieros sin conocimientos profundos de programación pudieran colaborar directamente sobre la interfaz, algo imposible en diseño 100% programático.

Aplicación en el proyecto Wee 3.0

En Wee 3.0, la adopción de esta técnica fue decisiva para mantener la organización del proyecto conforme crecía en funcionalidades. Se creó un storyboard principal donde vivían flujos críticos como el login, y múltiples Storyboard References enlazaban a flujos independientes como:

- Onboarding y registro de usuario
- Perfil del asegurado
- Sección de videos de bienestar
- Administración de pólizas y documentos

Este esquema facilitó el desarrollo paralelo, el aislamiento de bugs visuales, y permitió escalar a más pantallas sin comprometer la estabilidad del sistema de navegación.

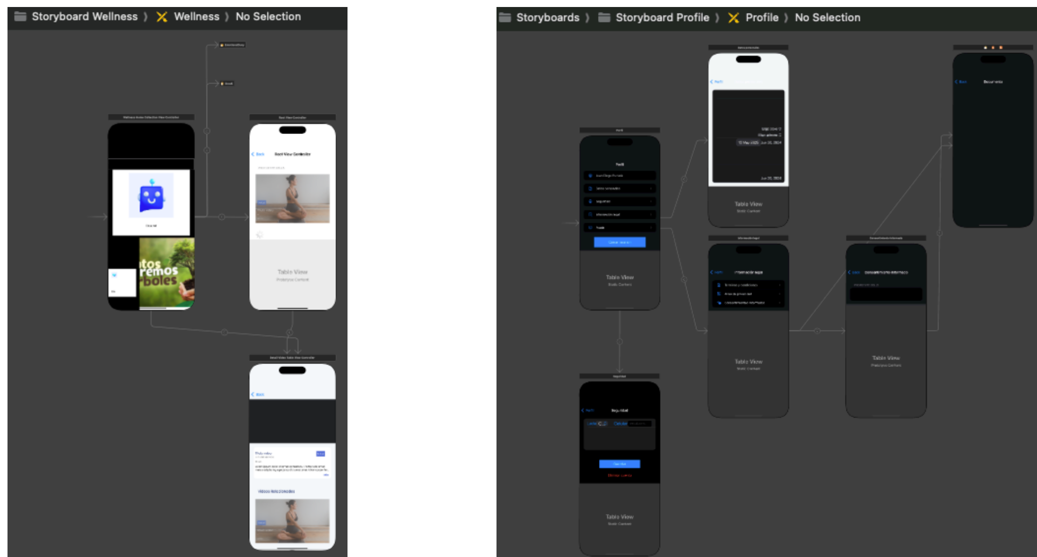


Figura 32. Storyboards independientes conectados mediante referencias visuales en Xcode

Nota: A la izquierda se muestra el *Storyboard Wellness*, encargado de la navegación entre el listado de videos, retos y secciones de bienestar emocional. A la derecha se encuentra el *Storyboard Profile*, que administra las vistas asociadas al perfil del usuario, edición de datos y configuración avanzada. Ambos storyboards funcionan de forma desacoplada en el proyecto, pero se conectan entre sí mediante *Storyboard References.*

Implementación de múltiples métodos de almacenamiento local

Durante la implementación del módulo de inicio de sesión, se diseñó una arquitectura orientada a la demostración, validación y comparación de distintas estrategias de almacenamiento local en iOS. Esto permitió evaluar su idoneidad técnica antes de su aplicación definitiva a otros módulos más sensibles dentro de la aplicación.

Se implementaron tres controladores independientes, cada uno responsable de encapsular un enfoque específico: UserDefaults, Keychain, y archivos plist mediante FileManager. Esta modularidad no solo permitió abstraer el acceso a los datos de sesión del usuario, sino que habilitó pruebas automatizadas e independientes para cada estrategia, garantizando su comportamiento sin acoplarlos al flujo de vistas.

- **UserDefaults:** Se empleó inicialmente para almacenar las credenciales del usuario como ejercicio práctico. Si bien su uso fue limitado exclusivamente a las primeras versiones del login con datos de ejemplo, permitió mostrar cómo serializar objetos simples y recuperarlos fácilmente al reiniciar la app. Sin embargo, debido a la falta de cifrado y su exposición directa, esta opción fue descartada en etapas posteriores por motivos de seguridad.

- **Keychain:** El controlador KeychainController fue diseñado para almacenar de manera segura información sensible como tokens de acceso y contraseñas. Utilizando la API de Keychain Services de Apple, se serializó un diccionario con los datos de sesión y se protegió mediante los atributos adecuados (kSecAttrAccessible, kSecAttrService, kSecAttrAccount). Esta solución garantiza persistencia tras reinstalación, así como cifrado por hardware del dispositivo.

- **Plist con FileManager:** A modo de demostración y respaldo técnico, se implementó un controlador adicional para almacenar datos en archivos plist internos. Esta estrategia se basó en la escritura directa de archivos codificados en el sistema de archivos de la aplicación, organizados por nombre y ubicados dentro del sandbox correspondiente. Se utilizó FileManager junto con PropertyListEncoder y Decoder para asegurar compatibilidad nativa y facilidad de inspección. Este método fue especialmente útil para almacenar información estructurada no sensible durante el proceso de desarrollo.

La arquitectura adoptada permite intercambiar fácilmente entre estos métodos en función del entorno (desarrollo o producción), gracias a la inyección de dependencias desde un UserSessionController superior. Este controlador coordina la lógica de almacenamiento, lo que a su vez facilita las pruebas unitarias y la adaptación a nuevas políticas de seguridad sin alterar el flujo de la aplicación.

```swift
/// Controlador para almacenamiento de sesión con UserDefaults.
/// Usado únicamente con datos de ejemplo en etapas tempranas del login.
final class UserSessionUserDefaultsController {

    /// Guarda el email actual del usuario.
    func saveEmail(_ email: String)

    /// Recupera el email almacenado.
    func retrieveEmail() -> String?

    /// Elimina cualquier dato relacionado con el usuario.
    func clear()
}
```

Figura 33. Firma del controlador de sesión UserSessionUserDefaultsController, diseñado para almacenar datos temporales de login utilizando UserDefaults de forma segura y controlada en etapas tempranas del flujo.

```swift
/// Controlador para almacenamiento estructurado con archivos plist.
final class PlistFileManagerController {

    /// Guarda un objeto codificable como plist en el sistema de archivos.
    func save<T: Codable>(_ object: T, to fileName: String) throws

    /// Recupera un objeto plist desde un archivo.
    func retrieve<T: Codable>(_ type: T.Type, from fileName: String) throws -> T?

    /// Elimina un archivo plist del sistema.
    func delete(fileName: String) throws
}
```

Figura 34. Firma del controlador PlistFileManagerController, encargado del almacenamiento estructurado mediante archivos plist, permitiendo guardar, recuperar y eliminar objetos codificables del sistema de archivos.

```swift
/// Controlador para almacenamiento seguro con Keychain.
final class KeychainController {

    /// Guarda datos codificados en el llavero para el email especificado.
    func save(_ data: Data, for email: String) throws

    /// Recupera datos del llavero usando el email como clave.
    func retrieve(for email: String) throws -> Data?

    /// Elimina datos del llavero para un email determinado.
    func delete(for email: String) throws
}
```

Figura 35. Firma del controlador KeychainController, utilizado para el almacenamiento seguro de datos codificados mediante Keychain, asociado a identificadores como el correo electrónico del usuario.

Peticiones de red seguras y manejo de errores

Seguridad en la conexión con URLSession

Durante la implementación del módulo de autenticación, se incorporó un mecanismo de cifrado de datos sensibles mediante el algoritmo RSA (Rivest–Shamir–Adleman), utilizando una clave pública proporcionada por el servidor. Esta decisión respondió a la necesidad de proteger credenciales y tokens del usuario desde el origen, incluso antes de que fueran transmitidos por la red, añadiendo una capa de seguridad adicional a la conexión HTTPS.

Para encapsular esta lógica, se diseñó una clase RSAController, responsable de gestionar el ciclo de vida de la clave pública y realizar el cifrado de cadenas de texto. Su diseño modular permite que cualquier otro módulo del sistema pueda reutilizarla sin conocer su implementación interna, respetando los principios de responsabilidad única y encapsulamiento.

```
final class RSAController {

    /// Cifra una cadena utilizando la clave pública RSA.
    func encrypt(_ string: String) throws -> String

    /// Establece una nueva clave pública proporcionada por el servidor.
    func setPublicKey(_ key: String)
}
```

Figura 36. Firma del controlador RSAController, empleado para establecer una clave pública RSA y cifrar datos sensibles mediante dicho esquema de encriptación.

La clase opera sobre datos tipo String, asegurando la compatibilidad con los flujos comunes de entrada de usuario y transmisión JSON. Antes de cada solicitud crítica (como login), la contraseña es cifrada usando este controlador, de forma que su representación cifrada viaje como parte del cuerpo de la petición.

Para fortalecer el control de errores, se definió un enumerador RSAError que agrupa los fallos más comunes durante el proceso de cifrado, como claves mal formateadas, errores al convertir cadenas a datos binarios, o fallos en el propio proceso criptográfico.

```
enum RSAError: Error {
    case invalidPublicKey
    case encryptionFailed
    case stringToDataConversionFailed
}
```

Figura 37. Enumeración RSAError para el manejo específico de errores en procesos de cifrado con RSA. Define casos como clave pública inválida, fallo de cifrado y error en la conversión de cadena a datos.

Este enfoque permitió una trazabilidad precisa y localizada de errores relacionados con el cifrado, mejorando el soporte en QA y facilitando el aislamiento de fallos durante pruebas integradas. Asimismo, al implementar este controlador como una unidad desacoplada, se sentaron las bases para su futura extensión o migración a tecnologías como Secure Enclave si se desease incrementar el nivel de protección a nivel de hardware.

Diseño genérico y controlado de respuestas de red

Una de las piezas arquitectónicas más sólidas y refinadas del proyecto Wee 3.0 fue la construcción de una capa de red personalizada, segura, extensible y orientada a la reutilización de componentes. Este módulo, centrado en la clase RequestController, fue diseñado para encapsular por completo la lógica de peticiones HTTP usando URLSession, tipado genérico adaptable con Codable, y un sistema propio de manejo de errores con semántica precisa, mediante la adopción del protocolo LocalizedError.

Controlador genérico con compatibilidad total con Codable

La clase RequestController permite realizar solicitudes POST, GET y PUT de manera genérica, recibiendo cualquier tipo de cuerpo que conforme a Encodable, y esperando una respuesta de tipo Decodable. Esto fue posible gracias a la declaración del método principal con parámetros tipados como genéricos:

```
func post<T: Encodable, U: Decodable>(
    body: T,
    endpoint: String,
    expecting: U.Type
) async -> Result<U, FetchError>
```

Figura 38. Firma de la función post genérica para realizar peticiones de red tipo POST. Utiliza tipos Encodable para el cuerpo de la solicitud y Decodable para la respuesta esperada, retornando un Result que encapsula éxito o error del tipo FetchError.

Con esta firma, el mismo método puede utilizarse para autenticar usuarios, enviar formularios o recuperar configuraciones, sin duplicar lógica ni comprometer el tipo de respuesta esperada. La conversión de objetos a Data y viceversa se realiza mediante JSONEncoder y JSONDecoder, lo cual garantiza compatibilidad total con los estándares RESTful modernos.

Uso de Result<T, Error> y propagación explícita de fallos

A diferencia del enfoque tradicional basado en try/catch, el método post() retorna un Result<U, FetchError>, encapsulando tanto la respuesta esperada como cualquier posible error de red, codificación o del servidor. Esto promueve un flujo de control explícito y funcional, donde los estados de éxito y fallo pueden ser evaluados directamente, simplificando las operaciones aguas abajo.

```swift
let result = await requestController.post(body: request, endpoint: url, expecting: AuthResponse.self)

switch result {
case .success(let response):
    // manejar respuesta
case .failure(let error):
    // mostrar alerta o logear
}
```

Figura 39. Manejo de una petición de red con tipado genérico y Result. Esta estructura permite desacoplar el cuerpo (body), el tipo esperado (expecting) y el endpoint, facilitando pruebas, mantenimiento y lectura del código.

Modelo de error personalizado con semántica descriptiva

Para mejorar la trazabilidad y legibilidad de fallos, se diseñó un enum especializado llamado FetchError, que implementa el protocolo LocalizedError. Cada caso del enumerador representa un tipo de error con contexto técnico claro: desde ausencia de conexión (noInternet) hasta errores de decodificación, falta de autorización o respuestas mal estructuradas del servidor.

```swift
enum FetchError: LocalizedError {
    case noInternet
    case invalidStatusCode(Int)
    case decodingError
    case timeout
    case custom(String)

    var errorDescription: String? {
        switch self {
        case .noInternet: return "No hay conexión a Internet."
        case .invalidStatusCode(let code): return "Error del servidor. Código: \(code)."
        case .decodingError: return "No se pudo interpretar la respuesta del servidor."
        case .timeout: return "La petición ha tardado demasiado."
        case .custom(let message): return message
        }
    }
}
```

Figura 40. Definición del enumerador FetchError con conformidad a LocalizedError. Esta estructura permite representar errores de red de forma clara, semántica y personalizada, facilitando tanto el manejo interno como la presentación al usuario.

Esta estructura permite generar alertas informativas y contextualizadas durante desarrollo (Dev) o pruebas (QA), manteniendo un mensaje técnico pero claro para el equipo. En entornos de producción, se puede sustituir fácilmente por mensajes genéricos o notificaciones silenciosas.

URLSession y su delegado para control avanzado de red

Para complementar la seguridad y la trazabilidad, se creó un objeto URLSessionDelegateHandler, responsable de interceptar eventos del sistema de red en tiempo real. Esto permitió monitorear respuestas de bajo nivel como errores de TLS, problemas de certificados, reintentos automáticos ante caídas momentáneas de red, y establecer políticas explícitas para eventos de conectividad.

```
final class URLSessionDelegateHandler: NSObject, URLSessionDelegate {
    func urlSession(
        _ session: URLSession,
        didReceive challenge: URLAuthenticationChallenge,
        completionHandler: @escaping (URLSession.AuthChallengeDisposition, URLCredential?) -> Void
    )
}
```

Figura 41. Definición de URLSessionDelegateHandler. Este delegado personalizado permite interceptar desafíos de autenticación durante conexiones seguras, habilitando validación de certificados o políticas avanzadas de seguridad en URLSession.

Al delegar estas responsabilidades, se consiguió instrumentar un URLSession completamente personalizado y seguro, capaz de adaptarse a distintos entornos sin modificar el controlador de red principal.

Separación de lógica con controladores auxiliares de errores

Esta interfaz muestra cómo el controlador se encarga de capturar errores del sistema (URLError, DecodingError, POSIXError), mapearlos a una representación interna uniforme (FetchError) y, en entornos de desarrollo, presentar una alerta informativa detallada para el programador o el equipo de QA.

El uso de withCheckedContinuation permite suspender la ejecución asincrónica hasta que el usuario interactúe con la alerta, retornando un Result<T, FetchError> junto con un indicador Bool sobre si se debe reintentar la operación, fomentando decisiones dinámicas desde la capa de vista.

Esta estructura desacoplada, elegante y robusta facilitó significativamente la validación de errores durante el desarrollo, elevando la calidad del sistema y mejorando la experiencia del equipo técnico en procesos

de debugging y prueba.

```
/// Controlador auxiliar para interpretar errores de red y mostrar alertas informativas en desarrollo.
struct RequestErrorController {

    /// Maneja el error de red y presenta una alerta si es necesario.
    /// - Parameters:
    ///   - error: El error técnico capturado (URLError, DecodingError, etc.).
    ///   - moduleError: Error contextual propio del módulo llamador.
    ///   - viewController: Vista actual para mostrar la alerta.
    /// - Returns: Resultado con el error traducido, y un booleano que indica si el usuario desea reintentar.
    static func handleRequestErrorAndShowAlert<T>(
        _ error: Error,
        moduleError: LocalizedError?,
        viewController: UIViewController?
    ) async -> (Result<T, FetchError>, Bool)

    /// Traduce errores genéricos del sistema a errores del dominio (`FetchError`).
    static func handleRequestError(_ error: Error) -> FetchError

    /// Evalúa si un error corresponde al cliente (tiempo, red, etc.).
    static func isClientError(_ error: Error) -> Bool

    /// Evalúa si un error proviene del servidor (SSL, DNS, caída).
    static func isServerError(_ error: Error) -> Bool
}
```

Figura 42. Firma de funciones del RequestErrorController. Este controlador centraliza la interpretación de errores de red y permite mostrar alertas personalizadas al usuario durante el desarrollo, facilitando el diagnóstico y manteniendo el principio de separación de responsabilidades.

Integración de Compositional Layout y Diffable Data Source

La introducción de UICollectionViewCompositionalLayout y UICollectionViewDiffableDataSource en UIKit representó un cambio radical en la forma de construir interfaces dinámicas y altamente personalizables. Estas tecnologías, implementadas a partir de iOS 13, resolvieron muchos de los desafíos históricos del uso de UICollectionView, permitiendo una construcción declarativa, flexible y escalable de layouts complejos y fuentes de datos altamente eficientes.

¿Qué es Compositional Layout?

UICollectionViewCompositionalLayout es un sistema de diseño modular que permite componer visualmente secciones con diferentes estructuras dentro de una misma colección. Cada sección puede tener su propio diseño independiente (layout section), con elementos (items), grupos (groups), y configuraciones de espaciado y scroll completamente personalizados. Esto habilita:

- Vistas de mosaico, listas, carruseles y diseños asimétricos, todo desde un solo componente.
- Interfaz adaptable a iPhone, iPad y Mac sin necesidad de múltiples layouts.
- Layouts declarativos y reutilizables definidos en código, sin necesidad de múltiples clases delegadas.

¿Qué es Diffable Data Source?

UICollectionViewDiffableDataSource reemplaza al tradicional UICollectionViewDataSource con un enfoque moderno basado en snapshots. Utiliza identificadores únicos y estructuras de datos inmutables que permiten:

- Actualización automática y animada del contenido sin necesidad de performBatchUpdates.
- Seguridad en concurrencia gracias a su diseño inmutable.
- Separación clara entre modelo y vista, simplificando el mantenimiento.

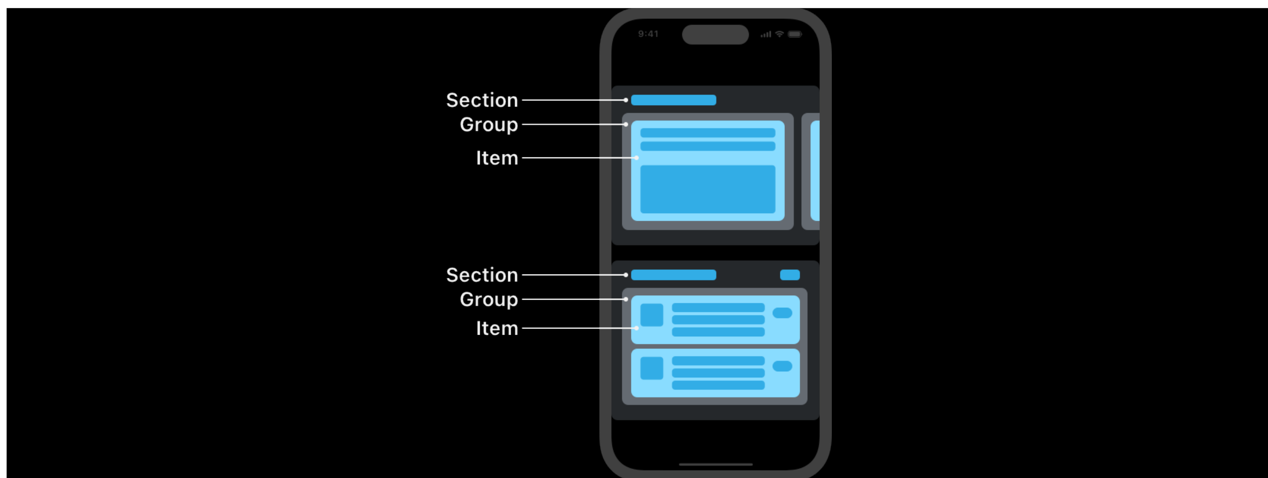


Figura 43. Composición visual de una interfaz con secciones heterogéneas mediante UICollectionViewCompositionalLayout

Nota: Esta imagen, tomada de la documentación oficial de Apple, ilustra cómo múltiples secciones pueden coexistir dentro de una misma UICollectionView, cada una con una estructura independiente definida por un NSCollectionLayoutSection.

Aplicación en el proyecto Wee 3.0

Durante el desarrollo de Wee 3.0, estas herramientas se integraron para construir flujos de contenido complejos como el de Bienestar emocional y Desafíos diarios, en los cuales:

- Cada sección representaba una categoría distinta de contenido, con un layout y comportamiento propio.

- Se usaron UICollectionViewCompositionalLayout para definir vistas jerárquicas con distintos tamaños de celdas y headers reutilizables.
- DiffableDataSource permitió actualizar el contenido en tiempo real tras consultas a la API, sin afectar el rendimiento ni la estructura visual.

Gracias a esto, fue posible diseñar interfaces multiplataforma, reutilizables y completamente desacopladas, facilitando pruebas, desarrollo paralelo y escalabilidad modular.

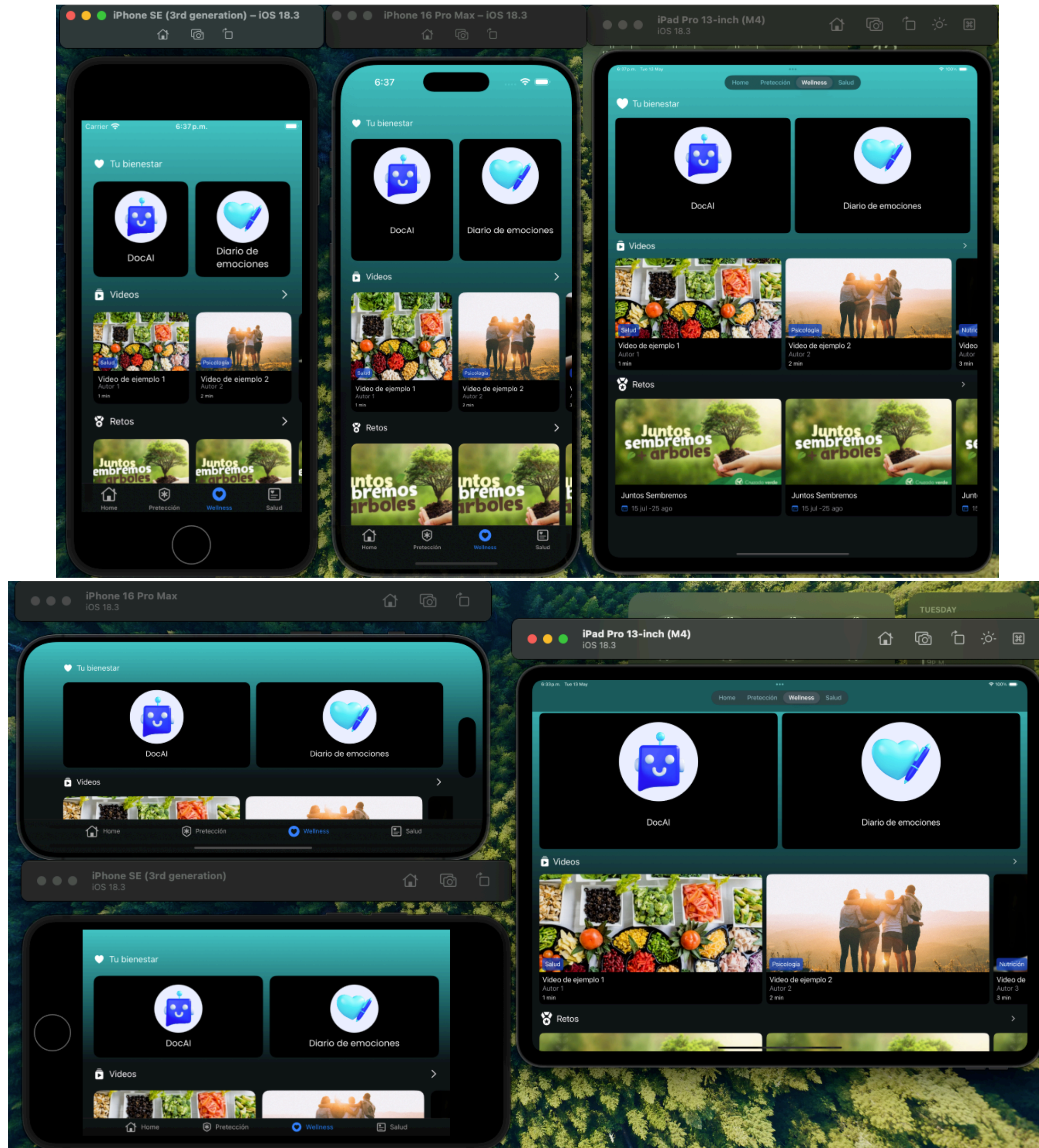


Figura 44. Diseño Modular con UICollectionViewCompositionalLayout

Nota: En esta imagen se ejemplifica la implementación de un diseño modular y adaptable utilizando UICollectionViewCompositionalLayout en conjunto con UICollectionViewDiffableDataSource. Se observa cómo, mediante el uso de layouts específicos para cada sección, se consigue que la interfaz se ajuste dinámicamente a diferentes dispositivos y orientaciones. Cada sección —representando temáticas como bienestar, videos y retos— dispone de un comportamiento de desplazamiento y dimensiones calculadas en función del entorno, lo que permite mantener una estructura visual coherente y escalable. Esta aproximación refuerza las ventajas de los Storyboards y Auto Layout en UIKit, facilitando el mantenimiento y la ampliación del código, en línea con las directrices de diseño de Apple.

Capítulo 4 – Cierre técnico y profesional del proyecto

Aplicación integral de conocimientos de ingeniería

A lo largo del proyecto, pude constatar cómo mi formación como ingeniero mecatrónico me permitió enfrentar los desafíos técnicos y organizativos con una perspectiva estructurada, analítica y orientada a soluciones sostenibles. Si bien el desarrollo de software no es el eje central de la carrera, la lógica de control, el pensamiento modular y el diseño de sistemas complejos forman parte esencial del perfil profesional, y fueron fundamentales para abordar la construcción de una aplicación robusta y flexible desde sus cimientos.

Más allá de los conocimientos técnicos, lo que más influyó fue la capacidad de análisis, la habilidad para descomponer un problema grande en componentes funcionales, y la toma de decisiones con base en criterios de eficiencia, mantenibilidad y escalabilidad. Estas competencias se reflejaron en la elección y estructuración de la arquitectura de la app, en la organización del trabajo por módulos, y en la definición de estrategias que facilitaran el mantenimiento a largo plazo.

Otro aspecto clave fue la comunicación del conocimiento. Durante el proyecto, asumí un rol activo en la capacitación del equipo, promoviendo una cultura de documentación clara y decisiones compartidas. Más que imponer soluciones, me enfoqué en generar entendimiento común, ofrecer contexto y crear herramientas que facilitaran la colaboración. Esta actitud fue clave para mantener la coherencia técnica del proyecto, especialmente considerando que la app debía escalarse y adaptarse al paso del tiempo y la integración de nuevos ingenieros al equipo.

Finalmente, esta experiencia me permitió integrar de forma natural lo aprendido en la carrera con lo exigido en un entorno profesional real. La capacidad de adaptación, el pensamiento sistémico, el trabajo en equipo y la responsabilidad técnica fueron elementos que me acompañaron desde los primeros sprints hasta la última entrega, confirmando que el enfoque ingenieril trasciende plataformas o lenguajes: se trata, en esencia, de resolver problemas complejos con soluciones estructuradas, claras y sostenibles.

Toma de decisiones tecnológicas y liderazgo técnico

Desde el inicio del proyecto tuve claro que una aplicación exitosa no solo se construye con herramientas modernas, sino con un equipo que entienda lo que hace, que se sienta cómodo con su entorno de desarrollo y que pueda crecer técnicamente a lo largo del camino. Por eso, muchas de las decisiones que tomé durante el diseño y la implementación no solo respondían a necesidades técnicas inmediatas, sino a

una visión a mediano plazo: formar un equipo sólido, capacitado y capaz de construir software verdaderamente nativo y sostenible.

El primer paso fue hacer un diagnóstico honesto de la situación inicial. Encontré una aplicación con problemas estructurales graves, falta de claridad en la arquitectura y un uso fragmentado de tecnologías. A partir de ese análisis, propuse una reestructuración completa del proyecto, donde cada decisión respondiera a principios de estabilidad, claridad y escalabilidad.

Opté por una arquitectura clara y conocida por el equipo, con el fin de reducir fricciones innecesarias y permitir que cada integrante pudiera enfocarse en construir y comprender, no solo en implementar. La prioridad era crear una base que permitiera avanzar de forma segura, y que sirviera como punto de partida para evolucionar hacia nuevas tecnologías, como SwiftUI, de manera gradual y natural, sin huecos en el conocimiento.

Además, impulsé activamente el uso de herramientas que facilitaran el trabajo en equipo, como la separación de entornos para pruebas y producción, la documentación clara, y una organización visual que permitiera entender los flujos sin necesidad de adentrarse en detalles técnicos. Todo esto con el objetivo de que el desarrollo fuera colaborativo, mantenible y alineado con una cultura de aprendizaje continuo.

Mi rol no fue solo técnico, sino también formativo. Asumí con responsabilidad la tarea de guiar, enseñar y proponer, creando un entorno donde el conocimiento se compartiera y se multiplicara. Esa fue, desde el principio, la base sobre la que debía construirse la aplicación: un equipo fuerte, unificado, que entiende lo que hace y por qué lo hace.

Crecimiento profesional en entorno real

Este proyecto representó una experiencia formativa profunda, no solo por los retos técnicos, sino por el entorno profesional en el que se desarrolló. El trabajo no se limitó al desarrollo de código, sino que exigió una coordinación constante con distintas áreas como QA, backend y producto. Participé activamente en las validaciones funcionales al cierre de cada sprint, colaborando estrechamente con el equipo de calidad y la Product Owner para asegurar que los entregables fueran funcionales, probados y alineados con los objetivos del negocio.

También me adapté rápidamente a un entorno de trabajo ágil, con entregas quincenales y una dinámica constante de revisión, priorización y redefinición de objetivos. Esto me permitió consolidar habilidades de planificación y seguimiento, así como aprender a mantener un ritmo de trabajo sostenible y realista, con metas claras en cada iteración.

Uno de los aspectos más valiosos de esta experiencia fue la autonomía con la que pude trabajar. Desde el principio asumí la responsabilidad técnica del proyecto, tomando decisiones críticas en cuanto a arquitectura, seguridad, patrones de diseño y organización del código, todo ello sin una supervisión directa constante. Esto implicó un compromiso alto con la calidad y la sostenibilidad del sistema.

En varios momentos fue necesario resolver problemas complejos bajo presión, ya fuera por fechas límite, por regresiones inesperadas o por requerimientos urgentes. Esta presión no fue un obstáculo, sino un catalizador para afianzar mi criterio técnico, fortalecer mi autonomía y mejorar mi capacidad para tomar decisiones informadas en tiempo real. En resumen, esta experiencia me permitió desarrollarme como profesional completo, con una visión técnica sólida y una capacidad real para liderar proyectos dentro de un equipo multidisciplinario.

Condiciones del cierre del proyecto

Mi salida del proyecto se dio en agosto de 2023, aproximadamente seis meses antes de la entrega formal estimada, la cual estaba planeada para principios de febrero de 2024. Aunque no presencié el despliegue final de la aplicación, entregué un MVP funcional con bases sólidas para su evolución y consolidación.

Al momento de mi salida, la aplicación aún no había sido liberada a usuarios reales, ya que la versión anterior —Wee 2.0— continuaba operando en producción para cumplir con compromisos previamente adquiridos con clientes. La nueva versión permanecía en etapa de desarrollo activo, resguardada en el repositorio privado de la empresa.

No se designó un responsable técnico único para continuar el proyecto, por lo cual enfoqué mi liderazgo en formar un equipo capacitado y autónomo. Esta estrategia aseguró que, ante mi ausencia o la de cualquier otro integrante, el equipo tuviera los conocimientos necesarios para continuar el desarrollo sin interrupciones.

Entregué recomendaciones claras para futuras decisiones técnicas, especialmente en torno a una eventual migración hacia SwiftUI. Expliqué que el uso inicial de UIKit permitiría comprender a fondo la arquitectura de información y los flujos internos de la app. Con esta base, el equipo podría planear una transición gradual hacia SwiftUI, con una visión clara de qué estructuras serían más eficientes bajo el nuevo paradigma declarativo.

Los módulos que dejé implementados —principalmente inicio de sesión, sección Wellness, DocAI y la arquitectura base de la aplicación— incluían una arquitectura desacoplada, modular, y extensible, comunicación robusta con el backend, manejo de errores avanzado, layouts adaptativos con UICollectionViewCompositionalLayout, y compatibilidad completa con iPhone, iPad y Mac (a través de Catalyst).

Si bien no se alcanzó a implementar CI/CD ni pruebas automatizadas formales, el código entregado era estable, limpio y documentado, con enfoque en calidad y buenas prácticas. Más allá de una entrega técnica, el legado del proyecto quedó reflejado en la estandarización del trabajo, la capacitación del equipo y la construcción de una base sólida para su evolución futura.

Bibliografía

1. Dirección General de Bibliotecas y Servicios Digitales de Información, UNAM. (s.f.). Cómo hacer citas y referencias en formato APA. https://bibliotecas.unam.mx/index.php/desarrollo-de-habilidades-informativas/como-hacer-citas-y-referencias-en-formato-apa

2. Diagnostikare. (s.f.). El primer paso para sentirte mejor. https://www.diagnostikare.com/

3. Endeavor Hub. (2022). El nuevo modelo de negocio de Betterfly. https://endeavor-hub.com/hub-article-estrategia-el-nuevo-modelo-de-negocio-de-betterfly/

4. ICEX. (2024). Healthtech en México 2024. https://www.icex.es/content/dam/es/icex/oficinas/077/documentos/2024/05/anexos/FS_HealthTech%20en%20M%C3%A9xico%202024_REV.pdf

5. MAPFRE. (2024). El sector insurtech latinoamericano cierra 2024 con más de 500 insurtech en un contexto adverso. https://www.mapfre.com/comunicacion/innovacion-comunicacion/insurtech-latinoamericano/

6. WeeCompany. (s.f.). Acerca de nosotros. https://www.weecompany.net/acerca_de.html

7. Apple. (s.f.-a). API Design Guidelines. Swift.org. Recuperado de https://www.swift.org/documentation/api-design-guidelines/

8. Apple. (s.f.-b). Organizing Files in Your Xcode Project. Apple Developer Documentation. Recuperado de https://developer.apple.com/documentation/xcode/organizing-your-project-s-files/

9. Apple. (2021a). Discover DocC documentation in Xcode. WWDC21 – Session 10166. Recuperado de https://developer.apple.com/videos/play/wwdc2021/10166/

10. Apple. (2021b). Create documentation in Xcode. WWDC21 – Session 10235. Recuperado de https://developer.apple.com/videos/play/wwdc2021/10235/

11. Apple. (2023a). What's new in Swift documentation. WWDC23 – Session 10244. Recuperado de https://developer.apple.com/videos/play/wwdc2023/10244/

12. Apple. (s.f.-c). Model-View-Controller. Apple Developer Documentation. Recuperado de https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html

13. Apple. (s.f.-d). View Controller Programming Guide for iOS. Apple Developer Documentation. Recuperado de https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/

14. Apple. (s.f.-e). About App Development with UIKit. Apple Developer Documentation. Recuperado de https://developer.apple.com/documentation/uikit/about-app-development-with-uikit

15. Apple. (s.f.-f). View Controllers. Apple Developer Documentation. Recuperado de https://developer.apple.com/documentation/uikit/view-controllers

16. Apple. (s.f.-g). View Controller Programming Guide for iOS. Apple Developer Documentation. Recuperado de https://developer.apple.com/library/archive/featuredarticles/ViewControllerPGforiPhoneOS/

17. Apple. (s.f.-i). Combine. Apple Developer Documentation. Recuperado de https://developer.apple.com/documentation/combine/

18. Apple. (s.f.-j). URLSession. Apple Developer Documentation. Recuperado de https://developer.apple.com/documentation/foundation/urlsession/

19. Apple. (2021c). Meet async/await in Swift. WWDC21 – Session 10132. Recuperado de https://developer.apple.com/videos/play/wwdc2021/10132/

20. Apple. (2021d). Explore structured concurrency in Swift. WWDC21 – Session 10095. Recuperado de https://developer.apple.com/videos/play/wwdc2021/10095/

21. Apple. (s.f.-k). UserDefaults. Apple Developer Documentation. Recuperado de https://developer.apple.com/documentation/foundation/userdefaults/

22. Apple. (s.f.-l). Keychain Services. Apple Developer Documentation. Recuperado de https://developer.apple.com/documentation/security/keychain-services/

23. Apple. (s.f.-m). FileManager. Apple Developer Documentation. Recuperado de https://developer.apple.com/documentation/foundation/filemanager/

24. Apple. (s.f.-n). Core Data. Apple Developer Documentation. Recuperado de https://developer.apple.com/documentation/coredata/

25. Apple. (s.f.-o). Views with Intrinsic Content Size. Apple Developer Documentation. Recuperado de https://developer.apple.com/library/archive/documentation/UserExperience/Conceptual/AutolayoutPG/ViewswithIntrinsicContentSize.html

26. Apple. (s.f.-p). UITabBarController. Apple Developer Documentation. Recuperado de https://developer.apple.com/documentation/uikit/uitabbarcontroller

27. Apple. (s.f.-q). UINavigationController. Apple Developer Documentation. Recuperado de https://developer.apple.com/documentation/uikit/uinavigationcontroller

28. Sergio Humberto. (s.f.). Conociendo GitFlow. Medium. Recuperado de https://medium.com/@sergiohumberto27/conociendo-gitflow-a588716fbc28

29. Hudson, P. (s.f.). How to use storyboard references to simplify your storyboards. Hacking with Swift. Recuperado de https://www.hackingwithswift.com/example-code/xcode/how-to-use-storyboard-references-to-simplify-your-storyboards

30. Apple. (s.f.-r). UICollectionViewCompositionalLayout. Apple Developer Documentation. Recuperado de https://developer.apple.com/documentation/uikit/uicollectionviewcompositionallayout

31. Apple. (s.f.-s). UICollectionViewDiffableDataSource. Apple Developer Documentation. Recuperado de https://developer.apple.com/documentation/uikit/uicollectionviewdiffabledatasource-9tqpa