Luis Alexander Peraza Corobo

líneas de productos software

# Introducción

En el contexto de la asignatura "Líneas de Productos Software", este documento recoge los resultados y aprendizajes adquiridos a lo largo de las prácticas realizadas durante el curso. La temática central se ha centrado en el desarrollo de aplicaciones móviles utilizando Xcode, Swift y SwiftUI para la plataforma iOS, herramientas fundamentales en la creación de productos software modernos y eficientes.

La primera sesión se dedicó a una introducción general a Xcode y los fundamentos de Swift y SwiftUI, proporcionando los conocimientos básicos necesarios para comenzar a desarrollar aplicaciones. En la segunda sesión, profundizamos en la combinación de vistas, explorando cómo se pueden integrar diferentes componentes de interfaz de usuario para crear experiencias más ricas e interactivas.

La tercera sesión estuvo enfocada en el manejo de listas y la navegación dentro de las aplicaciones, elementos clave para facilitar la interacción del usuario con el contenido. En la cuarta sesión, aprendimos sobre la gestión dinámica de listas y el uso de TabView, permitiendo a los desarrolladores construir interfaces más complejas y fluidas.

Finalmente, en la quinta sesión, nos adentramos en la persistencia de datos a través de CoreData, una herramienta esencial para gestionar y almacenar información de manera eficiente en nuestras aplicaciones.

Este documento sintetiza los conceptos y habilidades adquiridos en cada una de estas sesiones, ilustrando cómo se entrelazan para formar un conjunto cohesivo que sienta las bases del desarrollo de aplicaciones iOS. A través de ejemplos prácticos y reflexiones personales, se espera ofrecer una visión clara del proceso de aprendizaje y la aplicabilidad de estas herramientas en la creación de productos software de calidad.

# [Sesión 01 - Xcode, Swift, SwiftUI e iOS: Introducción](https://aulavirtual.ual.es/ultra/courses/_18132_1/outline/edit/document/_1158238_1?courseId=_18132_1&view=content)

Apple ha mantenido desde sus inicios un enfoque distintivo en el desarrollo de software, creando y manteniendo sus propias herramientas de desarrollo para garantizar una experiencia coherente y optimizada en sus plataformas. Este ecosistema cerrado permite a Apple mantener altos estándares de calidad y rendimiento, asegurando que las aplicaciones aprovechen al máximo las capacidades de sus dispositivos.

En el centro de este ecosistema se encuentra Xcode, el entorno de desarrollo integrado (IDE) oficial de Apple. Esta herramienta todo-en-uno proporciona a los desarrolladores todo lo necesario para crear aplicaciones para iOS, macOS, watchOS y tvOS. Complementando a Xcode, encontramos Swift Playgrounds, una innovadora herramienta que permite a desarrolladores experimentar con código de manera interactiva y aprender los fundamentos de la programación de forma práctica.

Antes del nacimiento de Swift, Objective-C era el lenguaje principal para el desarrollo en plataformas Apple. Como una extensión del lenguaje C, Objective-C introdujo capacidades de programación orientada a objetos mientras mantenía un enfoque imperativo, donde los desarrolladores debían especificar explícitamente cómo realizar cada tarea.

Sin embargo, en 2014, Apple presentó Swift como su nuevo lenguaje de programación moderno. Swift representó un cambio significativo en la filosofía de desarrollo, adoptando un enfoque más declarativo que permite a los desarrolladores expresar sus intenciones de manera más clara y concisa. Mientras que Objective-C requería una sintaxis más verbosa y propensa a errores, Swift introdujo una sintaxis moderna y expresiva que facilita la escritura y lectura del código.

Las mejoras que Swift aporta son sustanciales: mayor legibilidad gracias a una sintaxis más limpia y moderna, mejor calidad de código mediante un sistema de tipos más seguro y características como opcionales que previenen errores comunes, y mayor potencia con características avanzadas como protocolos, extensiones y programación funcional. Estas mejoras han resultado en aplicaciones más seguras, mantenibles y eficientes.

Foundation, como framework fundamental, proporciona las abstracciones básicas necesarias para el desarrollo en plataformas Apple. Esta biblioteca define los tipos de datos fundamentales, colecciones y otras utilidades esenciales que forman la base de cualquier aplicación. Su diseño cuidadoso asegura consistencia y eficiencia en las operaciones más básicas.

En cuanto a la interfaz de usuario, UIKit ha sido durante años el framework estándar para la creación de interfaces en iOS, siguiendo un modelo imperativo donde los desarrolladores deben gestionar explícitamente el estado y la actualización de la interfaz. Sin embargo, con la introducción de SwiftUI en 2019, Apple revolucionó el desarrollo de interfaces al adoptar un paradigma declarativo que simplifica enormemente la creación de interfaces de usuario. SwiftUI permite a los desarrolladores describir qué debe mostrar la interfaz en lugar de cómo construirla, resultando en código más conciso, mantenible y menos propenso a errores, además de facilitar la creación de interfaces consistentes a través de todas las plataformas de Apple.

Algunas características muy interesantes de Swift son:

1. Facilidad al trabajar con cadenas de caracteres: concadenar /(variable) para obtener el valor string
2. La posibilidad de marcar como opcionales variables por medio del operador “?”, es decir, su capacidad de recibir valores nulos y, por consecuencia, un operador para poder trabajar con ellos sin realizar ninguna comprobación por medio del operador (“?”)
3. El uso de clausaras para reducir la cantidad de código al introducir la lógica de una función en su propia llamada

En relación a las respuestas a las preguntas:

FOTO

Hecho esto, el siguiente fue construir nuestra primera interfaz gráfica. Para ello, configuramos el entorno para nuestro primer proyecto tal como ha sido documentado en el documento ofrecido por el profesor:

FOTO

Hecho esto, el siguiente paso fue introducir como funciona nuestro punto de entrada. Este define como vista principal el definido en ContentView(). Tras ello, vemos el contenido de ContentView() el que define un Text() con un espaciado interno (padding)

FOTO

Nótese ContentView\_Preview que define una previsualización de la vista hecha anteriormente. Dicha previsualización es vista en la zona de trabajo llamda canvas. Como bien sabemos XCode tiene las siguientes zonas:  
  
Zona 1: Navegador del proyecto (explorador de archivos)  
Zona 2: Editor  
Zona 3: Canvas (zona de previsualización)  
Zona 4: Inspectores 🡪 ¿?  
Zona 5: Estado y ejecución de la app en el simulador 🡪 ¿?

FOTOS

En canvas se muestra en ejecución el código con lo que puede ser un poco lento. Para ello es necesario compilar el proyecto:

FOTO

En el tema de previsualizaciones podemos modificarlos con los siguientes modificadores:

FOTO

De igual modo, podemos previsualizar varias vistas:  
  
FOTO

Vemos una previsualización usando otro dispositivo:  
  
FOTO

Hecha esta breve introducción, construimos formalmente la vista. La vista será simple. Lo primero que querremos será mostrar un mensaje personalizable recibido de algún modo. Mas concretamente querremos mostrar el nombre de la persona. Para ello usaremos una variable cuyo contenido será mostrado en un Text(). Para dar interactividad esta variable estará ligada al valor introducido en un TextEditor(). Para ello, necesia un tipo de variable especial. Dicha variable necesita trabajar con un property wrapper llamada @State que enlaza la variable local con cual interacción hecha en la UI para generar retroalimentación. Personalizando los componentes llegamos a lo siguiente:  
  
FOTO

Notarse que internamente la disposición que se sigue es verticial (tiene VStack interno)

# Sesión 02 - Xcode, Swift, SwiftUI e iOS: Combinación de vistas

En la segunda sesión de la asignatura, nos proponemos avanzar en el aprendizaje del lenguaje Swift y dar inicio al desarrollo de nuestra aplicación de gestión de amigos. Esta sesión se centra en la creación de una vista detalle, donde exploraremos el uso de stacks como contenedores para organizar componentes visuales, además de incorporar imágenes, símbolos y elementos de otros frameworks, como las vistas de mapa de UIKit.

La aplicación, denominada GestionAmigos, incluirá una vista principal llamada VistaDetalle, que se compondrá de varias sub-vistas: VistaMapa, VistaImage y VistaDatos. A lo largo de esta sesión, combinaremos diferentes vistas creadas de forma independiente, siguiendo la lógica aprendida en la primera sesión, para lograr un diseño cohesivo y funcional. Antes de entrar en la implementación, abordemos nuevos elementos de programación en Swift para enriquecer nuestras habilidades y preparación para el desarrollo.

En primer lugar, estudiemos cómo las funciones pueden ser utilizadas como parámetros, lo que permite mayor flexibilidad y modularidad en el código. Más concretamente estudiaremos el concepto de *closure*. Un *closure* es un elemento fundamental en Swift, ya que permiten encapsular bloques de código que se pueden ejecutar posteriormente. Mostremos un ejemplo para ilustrar dicho concepto con más detalles. Para ello, definamos una función que tome un valor entero y dos closures como parámetros. Estos closures representan acciones que se aplican a cada número en un rango determinado. La función iterará desde 1 hasta el valor especificado, imprimiendo los resultados de ambas acciones para cada iteración.

La llamada a la función la realizaremos de dos maneras: utilizando la notación clásica, donde los closures se definen en el mismo lugar de la llamada, y utilizando la notación de *trailing closure*, que simplifica la sintaxis al permitir que el primer closure se coloque fuera de los paréntesis de la función.

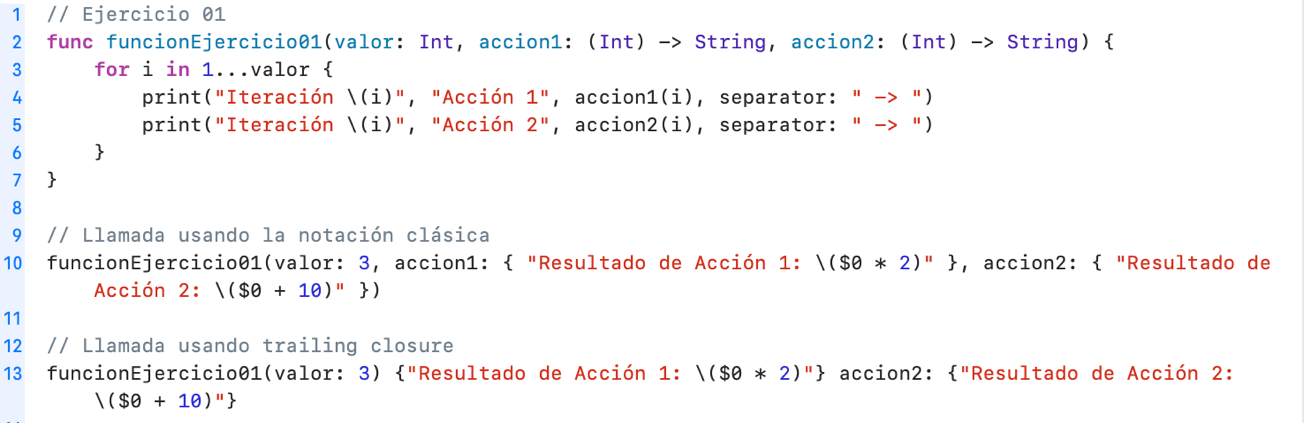


Figura 01: Ejercicio 01.

En segundo lugar, veamos cómo las funciones pueden interactuar con closures para generar resultados dinámicos basados en condiciones específicas. Para ello, crearemos una función *getMessage*, que toma un string y un closure que acepta un entero y un string como parámetros. La función genera un número aleatorio entre 1 y 3, y pasa este valor junto con el lugar especificado al closure. Dependiendo del número generado, el closure retorna un mensaje diferente que se utiliza para componer un mensaje final. La combinación de la generación de números aleatorios y el uso de closures en este contexto demuestra cómo se puede crear un comportamiento más interactivo y variado en nuestras aplicaciones

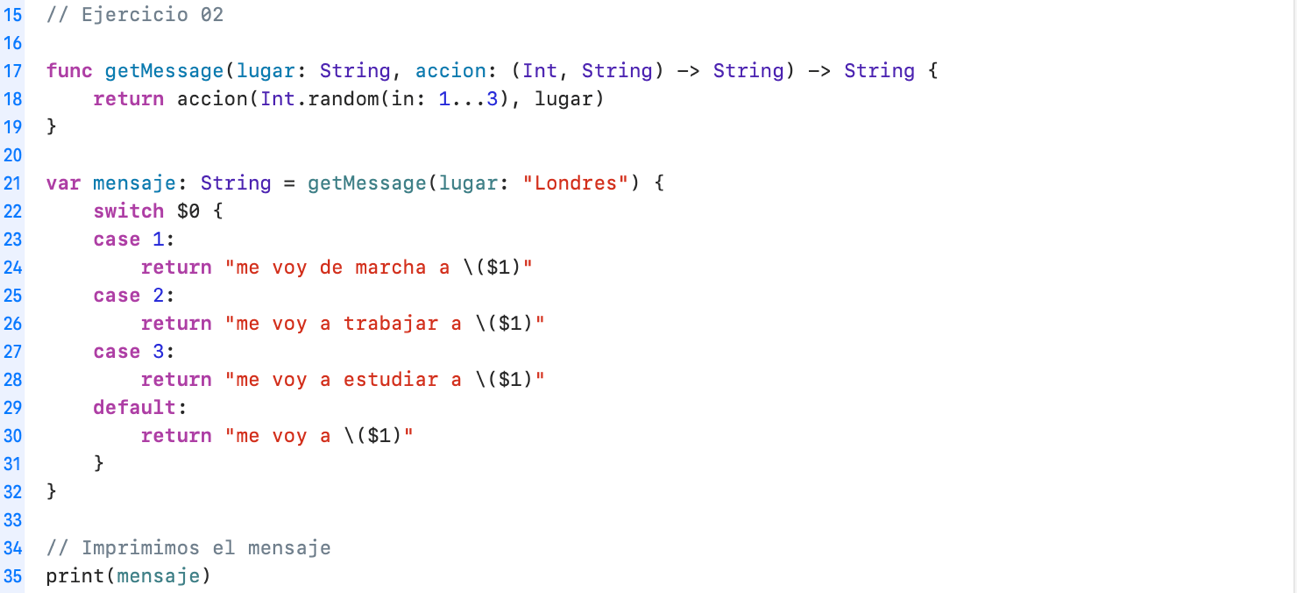


Figura 02: Ejercicio 02.

Veamos el resultado de ejecutar el código:

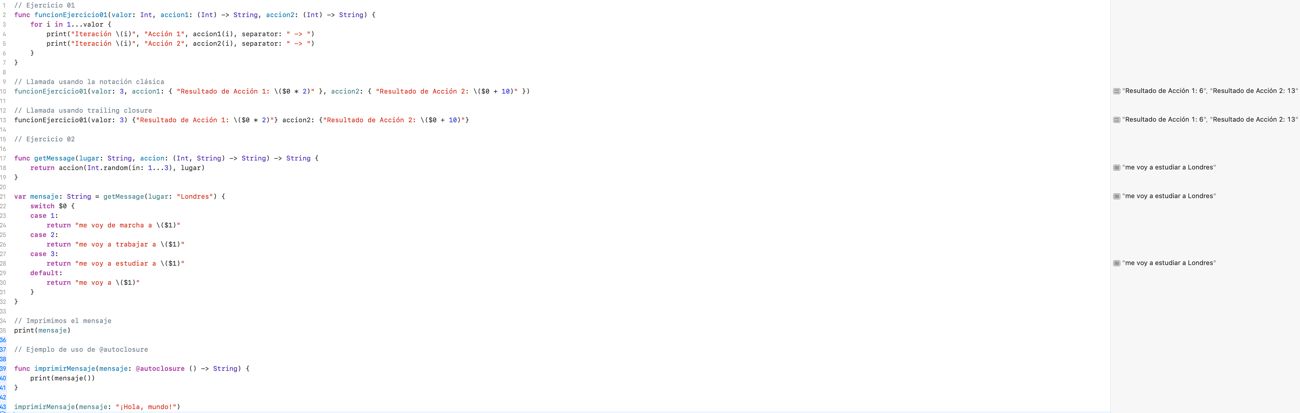


Figura 03: Ejercicio 01 y Ejercicio 02 - Salida.

Para terminar de profundizar nuestros conocimientos acerca de *closures* estudiemos dos conceptos fuertemente relacionados con ellas. Hablamos ni más ni menos que del uso de las etiquetas *@escaping* y *@autoclosure* en el proceso de declaración de las *clausures*. Por un lado, cuando se declara un *clausure* como *@escaping*, se indica que esta puede ser ejecutada después de que la función que la recibió haya terminado. Esto es útil en callbacks o en operaciones que requieren un tiempo de ejecución prolongado. Por otro lado, cuando una función toma un parámetro de tipo *@autoclosure*, el compilador convierte automáticamente la expresión proporcionada en una clausura que no requiere argumentos. Esto facilita la escritura de código más limpio y legible.

Dejando de lado el concepto de *closure*, sigamos profundizando en nuestro conocimiento sobre programación en Swift al hablar de un tema de igual o mayor importancia: la gestión de memoria. La forma en que Swift administra los recursos de hardware no solo afecta el rendimiento, sino también la estabilidad y escalabilidad de las aplicaciones. Para entender este proceso, es necesario explorar los enfoques de gestión estática y dinámica, y cómo estos influyen en la manipulación de datos y el comportamiento general del programa.

Por un lado, la gestión estática implica la asignación de memoria durante el tiempo de compilación. En este enfoque, la memoria se gestiona en la pila (*stack*), lo que lo hace rápido y eficiente. Este método es ideal para variables locales, ya que la memoria se libera automáticamente al salir del contexto de una función. Sin embargo, su flexibilidad es limitada porque el tamaño de las estructuras y variables debe ser conocido antes de la ejecución.

En lenguajes como Swift, las estructuras (*structs*), como el ejemplo de Point2D, se gestionan en la pila. Este enfoque asegura que cada instancia sea independiente, dado que las estructuras son tipos de valor. Al asignar una estructura a otra variable, se realiza una copia completa, lo que significa que los cambios en una instancia no afectan a las demás.

Por otro lado, la gestión dinámica se realiza durante el tiempo de ejecución utilizando el montón (*heap*). Este enfoque ofrece mayor flexibilidad, ya que permite crear estructuras de datos cuyo tamaño no necesita ser conocido de antemano. Sin embargo, requiere una gestión manual más cuidadosa para evitar problemas como fugas de memoria y errores en la desreferenciación de punteros nulos.

En Swift, las clases (*classes*) se gestionan en el montón y son tipos de referencia. Esto permite que múltiples referencias apunten al mismo objeto en memoria. Aunque este comportamiento facilita la creación de modelos compartidos, también implica que los cambios realizados a través de una referencia se reflejan en todas las demás, lo que puede dar lugar a efectos secundarios no deseados si no se gestionan adecuadamente.

En lenguajes como C, el concepto de gestión dinámica es realizada por medio de punteros, los cuales permiten manipular directamente direcciones de memoria, otorgando un control detallado pero también conlleva riesgos significativos. Por ejemplo, la desreferenciación de un puntero nulo puede causar errores graves y comprometer la estabilidad del programa. Aunque Swift gestiona automáticamente muchos de estos riesgos, es crucial entender estos conceptos para trabajar eficientemente en contextos de bajo nivel o interoperabilidad con otros lenguajes.

Con respecto al ejercicio de Point2D, se trata de un ejercicio que ilustra claramente la diferencia en el comportamiento de estructuras y clases en Swift. En nuestro caso Point2D es una estructura, entonces cada vez que se asigna a una nueva variable, se genera una copia independiente. En otras palabras, si una variable de tipo Point2D y hace cambios sobre esta, sus cambios solo se verían reflejados dentro si misma, pero no para la otra variable, pues es una copia independiente a la original. Esto no ocurriría si Point2D fuese una clase. En dicho caso, ambas variables apuntarían a la misma dirección de memoria y un cambio hecha por una se vería reflejado en la otra.

Realizada las explicaciones anteriores procederemos en realizar la implementación de nuestra aplicación. Como habíamos comentado la aplicación se componía de una vista principal llamada vista detalle que a su vez se componía de tres vistas: vista mapa, vista imagen y vista data. Tal que vista detalle es la suma de las diferentes vistas dichas anteriormente, pues nada mejor que empezar detallando el proceso de implementación de cada una de sus vistas. Indiferentemente de la vista creada la creación de su correspondiente archivo sigue el siguiente flujo:

1. Sobre la cabecera de XCode: file 🡪 new 🡪 file from template
2. Situarse sobre SwiftUI View y hace doble click

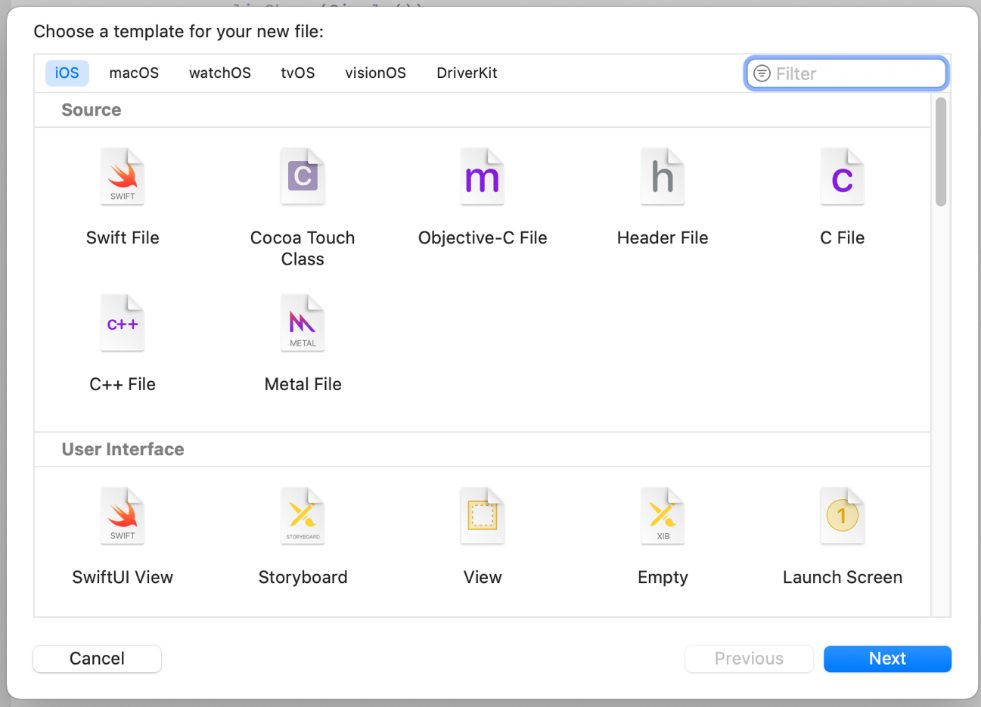


Figura 04.

1. Escribir un nombre para el archivo y seleccionar ubicación

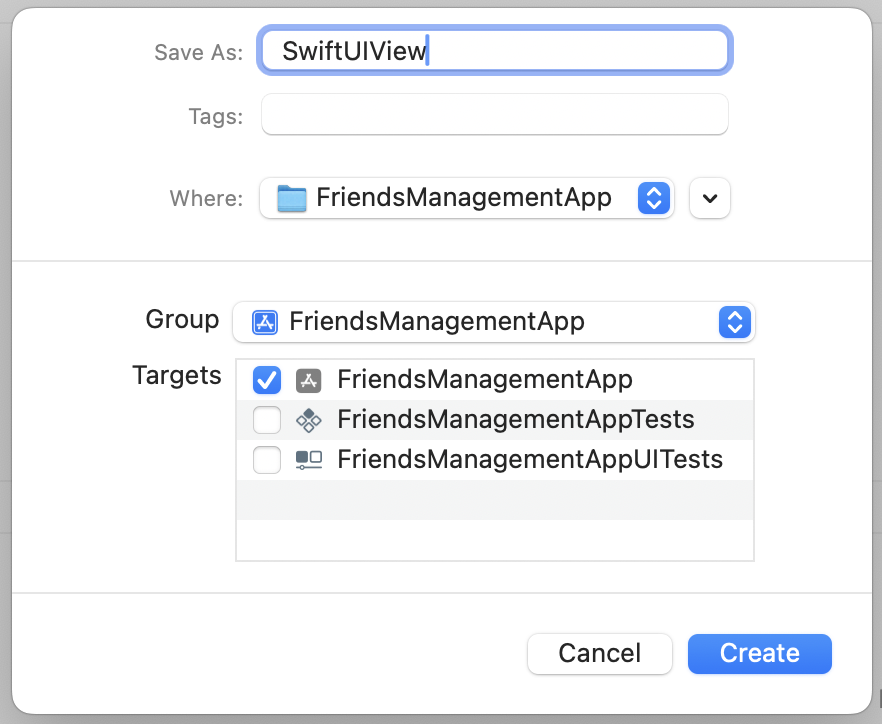


Figura 05.

Dicho esto, procedamos en crear el archivo de las tres vistas que componen a vista detalle. En primer lugar, tenemos vista mapa donde por medio de su código obtendremos una interfaz sencilla y efectiva para visualizar un mapa en una aplicación de iOS. El código utiliza el *framework MapKit* para mostrar un mapa centrado en unas coordenadas específicas (36.8305 de latitud y -2.4060 de longitud). Utiliza una propiedad de estado *@State* llamada región para gestionar la región visible del mapa, que está definida por un MKCoordinateRegion que incluye tanto el centro como el área visible (controlada por un MKCoordinateSpan). En el cuerpo de la vista, se crea un mapa vinculado a esta región mediante el uso del símbolo $, que permite que cualquier cambio en la región actualice automáticamente el mapa. Además, encapsula al mapa dentro de frame de altura de 250 puntos y que debe ignorar las áreas seguras del dispositivo, lo que le permite extenderse completamente hasta los bordes de la pantalla.

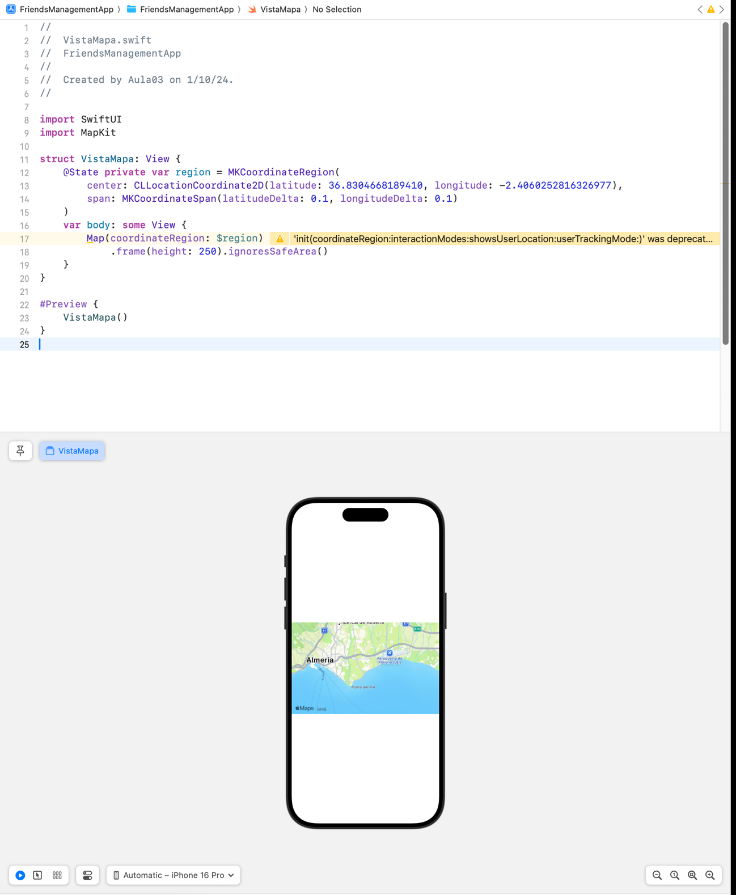


Figura 06. Vista mapa.

En segundo lugar tenemos vista imagen. Dicha vista presenta una imagen circular de, en este caso, Lionel Messi, y permite al usuario interactuar con ella para modificar su apariencia. Utiliza propiedades de estado como shadowColor, que define el color de la sombra y se inicializa en rojo; backgroundColor, que comienza como transparente; y borderWidth, que se establece en 2 puntos para el grosor del borde. En el cuerpo de la vista, la imagen se carga, se hace redimensionable y se establece en un tamaño de 150x150 puntos. Se le aplica un recorte en forma de círculo, un borde blanco y una sombra utilizando el color de sombra definido. Además, se desplaza la imagen 200 puntos hacia arriba. Cuando el usuario toca la imagen, la sombra, el fondo y el ancho del borde son cambiados. Para cambiar la sombra y el fondo definimos una función llamada randomColor que genera colores aleatorios mediante valores para los componentes rojo, verde y azul. El valor generado para el nuevo ancho de borde es un valor aleatorio entre 1 y 5 puntos.

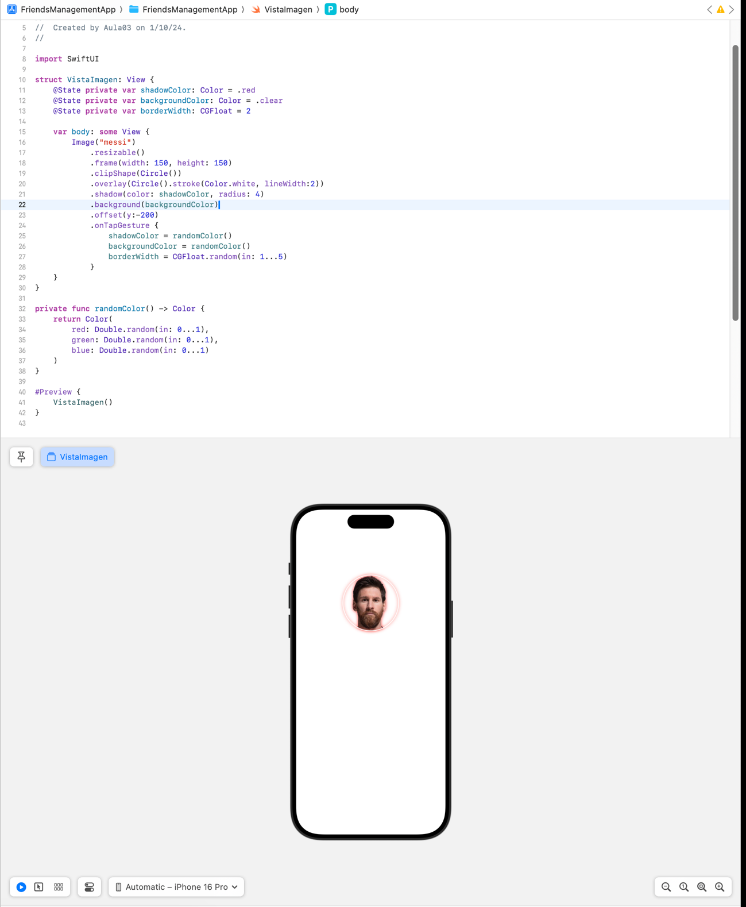


Figura 07. Vista imagen.

Por último, tenemos vista datos. Su código presenta información de una persona, en nuestro caso, Lionel Messi, de manera estructurada y visualmente atractiva. En el cuerpo de la vista, se utiliza un contenedor vertical (VStack) para organizar los elementos uno encima de otro. Al principio, se añade un espaciador con una altura de 80 puntos, lo que crea un margen superior. Luego, se define un Text para indicar el nombre de la persona. Se usa tamaño de fuente grande y color blanco. A continuación, se presenta un Label. Dicho Label utiliza un icono de un teléfono al cual concadena el correspondiente numero telefónico de la persona, usando un tamaño de fuente estándar. Se incluye un enlace (Link) que dirige al correo electrónico de la persona, acompañado de cierto icono y el texto correspondiente al enlace. Después, se agrega un divisor para separar visualmente las secciones. Se introduce una sección “About ‘nombre de la persona’”, con un subtítulo en un tamaño de fuente más pequeño que describe brevemente a la persona. Finalmente, se incorpora un espaciador al final para equilibrar la vista. El fondo de la vista se establece en un color verde, creando un contraste llamativo con el texto. Para utilizar los iconos se tuvo la necesidad de descargar la biblioteca de símbolos SF Symbols.

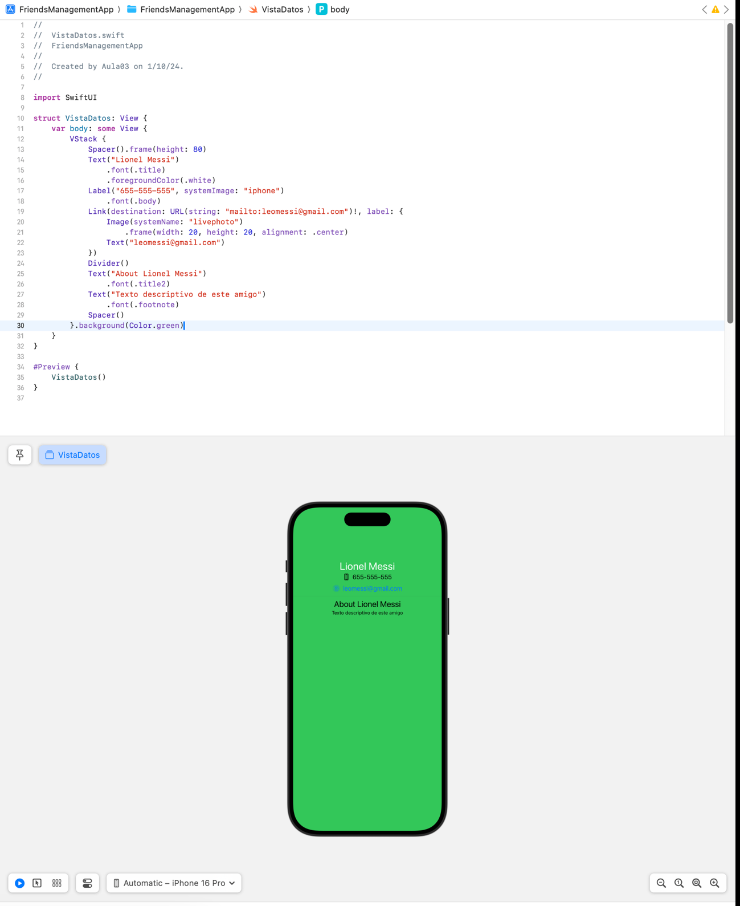


Figura 08. Vista datos.

Una vez vez explicada la implementación de las vistas que conforman a vista detalle, procedamos a juntarlas para armar e implementar la vista principal, también llamada vista detalle. Tal como vemos el código organiza y presenta las vistas previamente creadas. Para ello, primeramente, utiliza un contenedor en forma de pila (ZStack) para superponer elementos. Dentro de este ZStack, se coloca un VStack que contiene dos vistas: VistaMapa y VistaDatos. Así pues, conseguimos que vista imagen se superponga sobre el VStack que organiza vista mapa y vista datos uno encima de otro. Para ajustar la disposición visual, VistaDatos se desplaza hacia arriba en el eje vertical mediante el modificador .offset(y: -62).

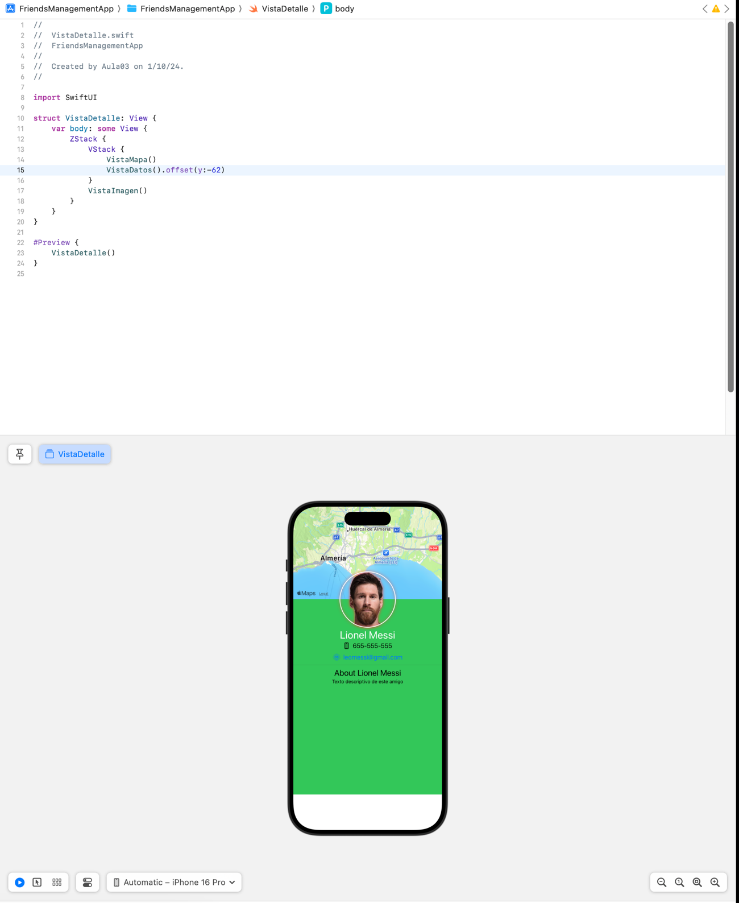
+

Figura 09. Vista detalle.

# Sesión 03 - Xcode, Swift, SwiftUI e iOS: Listas y Navegación

En la tercera sesión de la asignatura el objetivo será ampliar la funcionalidad de la aplicación GestionAmigos, que previamente incorporó la VistaDetalle para mostrar la información de un amigo específico. Más concretamente implementaremos una lista estática de amigos mediante una vista de tabla que presentará un resumen de cada uno, incluyendo un enlace a su detalle. Además, permitiremos a los usuarios no sólo navegar entre la lista y los detalles, sino filtrar amigos por el atributo favorito que definiremos, así como modificar su estado, además de dejar una opinión sobre cada persona, con ciertas restricciones. Para lograr esto, hemos seguido los siguientes pasos:

1. Crear un modelo de datos en forma de array estático
2. Desarrollar la vista de tabla
3. Establecer la navegación
4. Permitir la modificación del conjunto de datos.

De cara a la gestión de la lista de amigos, necesitaremos un modelo de datos que será el que se le pasará a las diferentes vistas para su gestión. Para separar el modelo de datos de las vistas, creamos un grupo llamado *Datos*. Para crear un grupo en XCode basta con situarse sobre el nombre del proyecto, botón derecho y presionar sobre New Group. Dentro de este grupo, tendremos dos archivos: el primero será la definición de la entidad *Amigo*, y el segundo, la definición de la estructura de datos *arrAmigos*, que será un array que contendrá a los amigos. Así pues, detallemos la creación de dichos archivos. En primer lugar tenemos al archivo correspondiente a la entidad *Amigo*. Como veremos en la siguiente fotografía el código presentado define una estructura (*struct*) la cual implementa el protocolo *Identifiable*. Este protocolo exige que cualquier tipo que lo adopte disponga de una propiedad id para identificar de manera única cada instancia. En este caso, el identificador único (id) se genera utilizando *UUID().uuidString*, lo que garantiza la unicidad de cada instancia creada. Los atributos de esta estructura encapsulan información básica relacionada con un amigo.

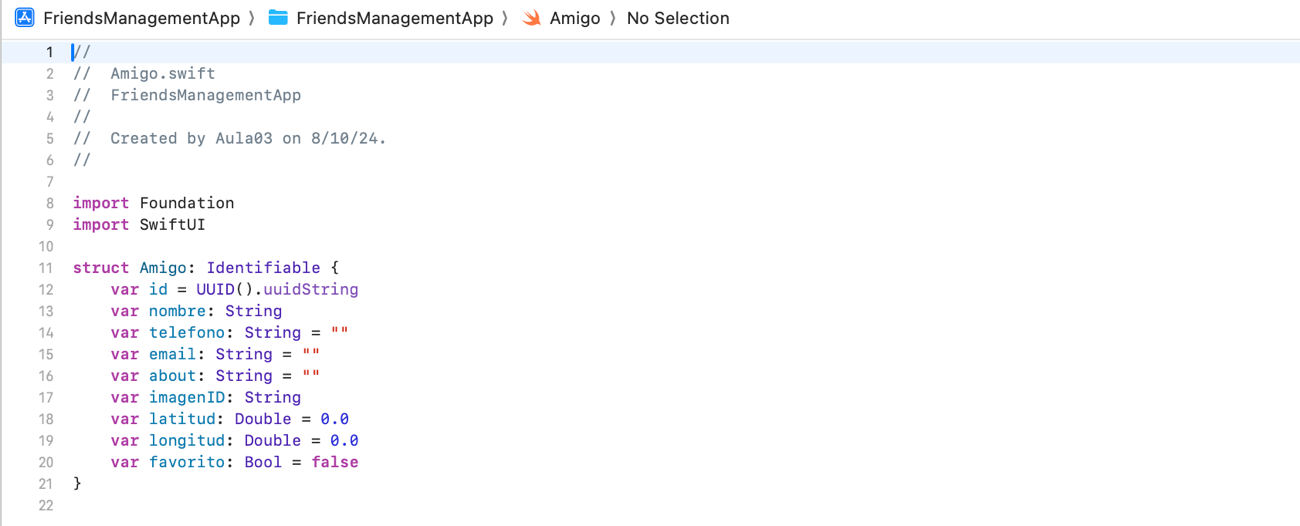


Figura 10. Entidad *Amigo*.

En segundo lugar tenemos al archivo dedicado a la gestión del modelo de datos, específicamente diseñado para encapsular la estructura de datos *arrAmigos*. Este archivo define una clase denominada *ModeloDatos*, la cual se ha marcado como *final* para impedir su herencia, garantizando así la integridad del modelo. El uso de una clase es fundamental en este contexto, ya que al ser un tipo por referencia, asegura que todas las vistas trabajen sobre los mismos datos, evitando inconsistencias que podrían surgir si se trabajara con copias. La clase implementa el protocolo *ObservableObject*, una característica fundamental de *SwiftUI* que permite establecer un sistema de observación y reacción automática ante cambios en las propiedades del modelo a través del publisher *objectWillChange*. La única y principal propiedad de esta clase es *arrAmigos*, un array que almacena objetos de tipo *Amigo*, y está decorada con el atributo *@Published*. Este *property wrapper* se encarga automáticamente de emitir notificaciones cuando el valor de la propiedad cambia, permitiendo que cualquier vista de *SwiftUI* que observe esta propiedad específica se actualice automáticamente para reflejar estos cambios. Su uso automatiza el proceso de envío de notificaciones, agilizando el proceso de desarrollo y creando un código más legible. Para proporcionar datos iniciales y facilitar las pruebas de la aplicación, se ha incluido una inicialización del array con cuatro instancias predefinidas que representan perfiles de usuarios con sus respectivas propiedades completamente configuradas.

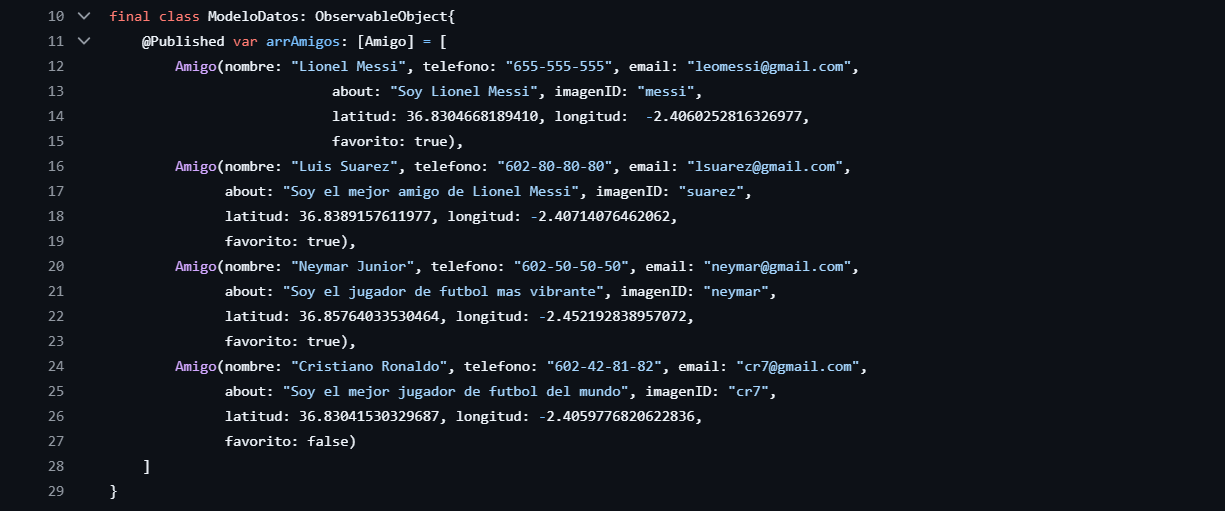
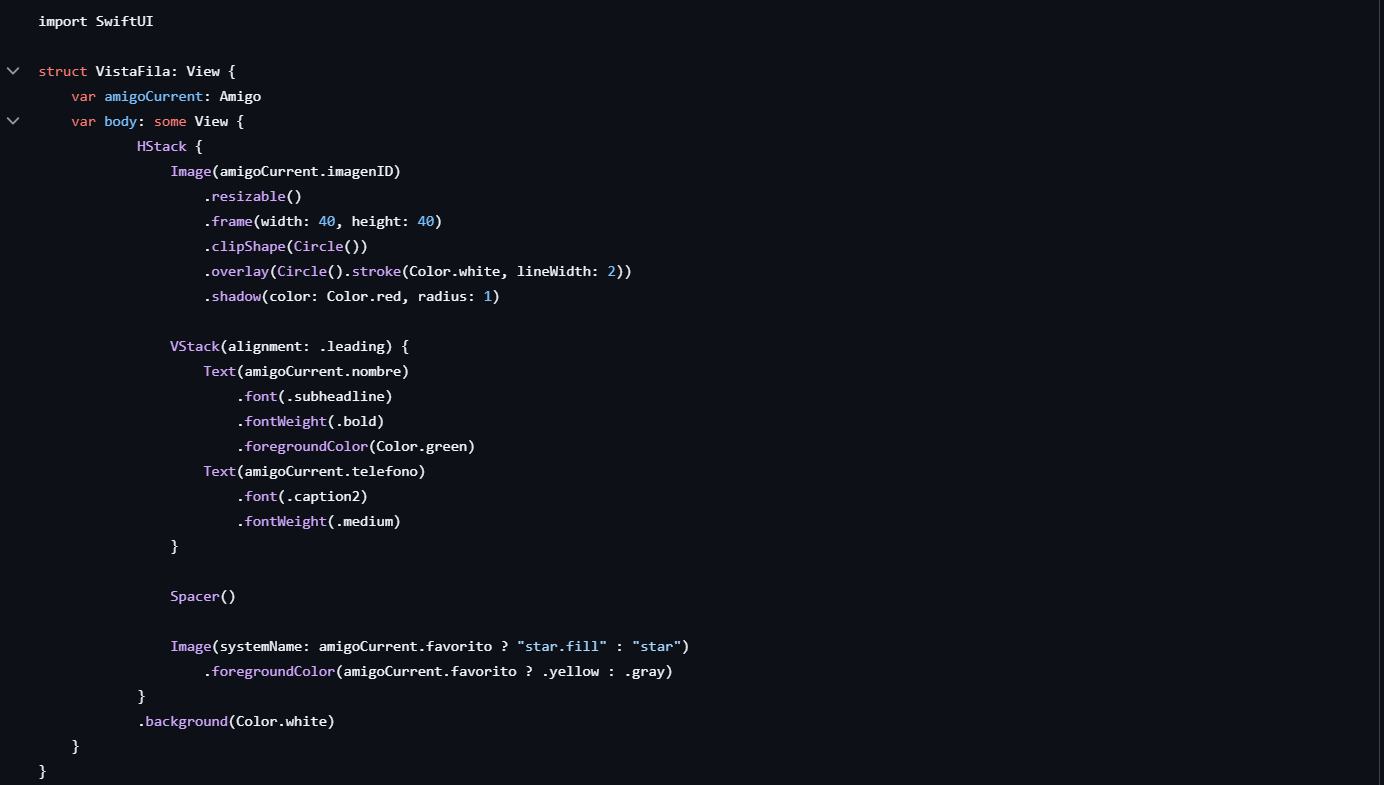


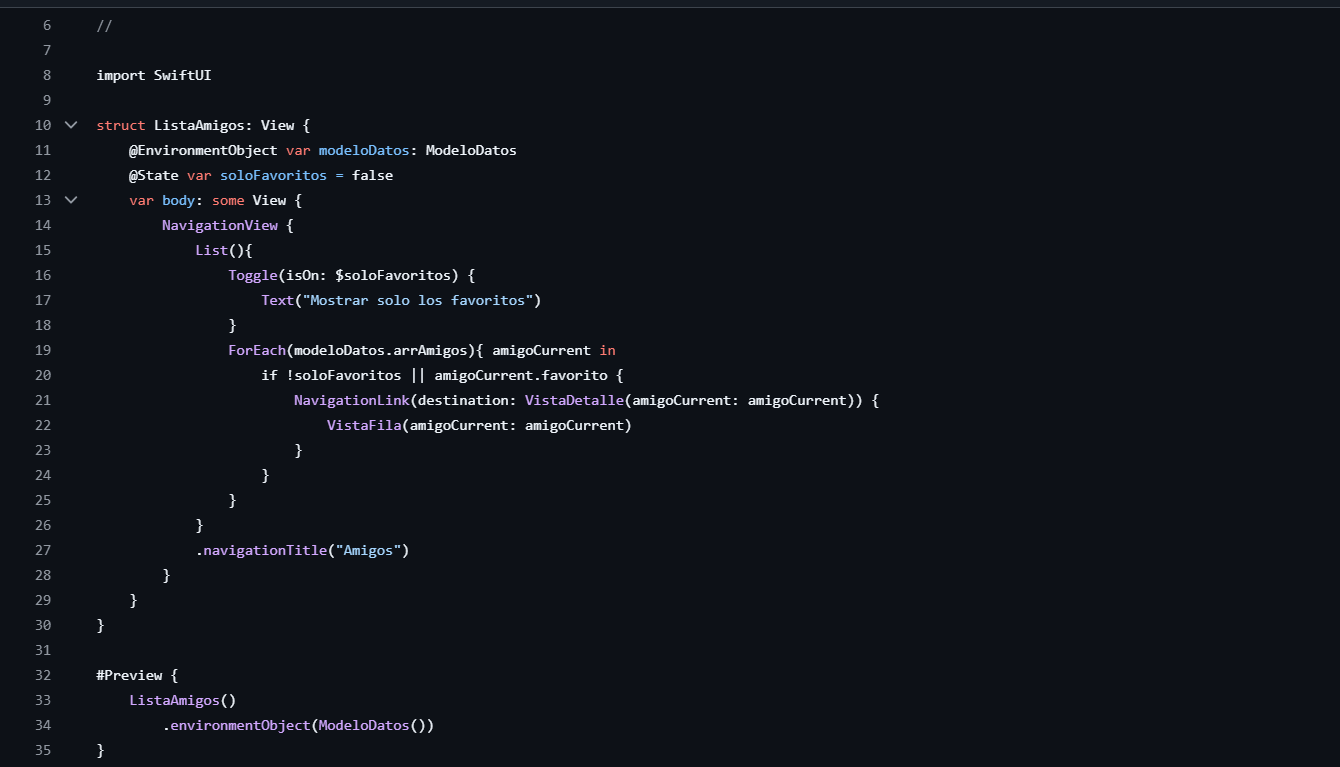
Figura 11. Estructura de datos *arrAmigos* (Cambiar foto).

Hecho el trabajo anterior, el siguiente paso será enlazar dicho modelo de datos en la vista que llamaremos *VistaListaAmigo*. Sin embargo, previo a su definición, crearemos una vista auxiliar llamada *VistaFila* que se encargará de renderizar cada elemento individual de nuestra lista de amigos. Esta vista está diseñada para recibir como parámetro un objeto de tipo Amigo llamado amigoCurrent, a partir del cual construirá la representación visual de una fila. La estructura de la fila se compone de un HStack (contenedor horizontal) con tres elementos principales. En el extremo izquierdo, se coloca una imagen del amigo que se redimensiona a 40x40 píxeles mediante .frame(width: 40, height: 40), se recorta en forma circular usando .clipShape(Circle()), se decora con un borde blanco de 2 píxeles de grosor y se realza con una sombra de color rojo. En la parte central, se implementa un VStack (contenedor vertical) alineado a la izquierda que muestra el nombre del amigo en color verde, con estilo de fuente subheadline y peso bold, seguido del número de teléfono con un tamaño de fuente caption2 y peso medium. En el extremo derecho, tras un Spacer() que empuja los elementos hacia los extremos, se coloca un icono de estrella que alterna entre lleno (star.fill) y vacío (star) según el estado de la propiedad favorito, cambiando su color entre amarillo (favorito) y gris (no favorito). Todo el contenido se presenta sobre un fondo blanco. Para facilitar el desarrollo y las pruebas, se ha incluido una vista previa que utiliza un ForEach para iterar sobre un array de amigos de prueba proporcionado por una instancia de ModeloDatos(), mostrando cómo se visualizará cada fila en la interfaz





Hecho esto, pasemos a construir VistaListaAmigos, la cual hemos llamado ListaAmigos. En primer lugar, la vista deberá recibir nuestro modelo de datos para trabajar sobre él. Para ello, definimos nuestro modelo de datos junto al property wrapper @EnvironmentObject. Esta decisión está relacionada con la estructura general de la aplicación: más adelante veremos que el punto de entrada FriendsManagementAppApp define un @StateObject para inicializar de manera única el modelo de datos, y establece ListaAmigos como la vista raíz de la aplicación dentro de una WindowGroup. La importancia de usar @EnvironmentObject radica en su capacidad para inyectar y acceder al modelo de datos desde cualquier vista en el árbol de vistas hijo de manera implícita, creando un código más limpio y legible. Sin este property wrapper, vistas como VistaDetalle, que más adelante mostraremos, deberían recibir el modelo de datos como parámetro explícito. Como curiosidad, debemos recordar que todo property wrapper es una definición de código en sí misma que añade un comportamiento especial sobre la propiedad en la cual se define. Hecho esto y con motivo de añadir el componente que permita filtrar la vista, mostrando únicamente el conjunto de amigos que el usuario ha declarado como favoritos (con su estrella amarilla), definiremos una variable de estado (@State) llamada soloFavoritos que rastree el valor booleano que emplearemos dentro del componente Toggle(), usado para realizar dicho filtrado. Como bien recordamos, el @State es una etiqueta colocada sobre una propiedad de vistas en SwiftUI que permite realizar seguimiento sobre cualquier cambio en la propiedad, tal que cuando cambia su valor, la vista se vuelve a renderizar automáticamente. Hecho esto, construimos la interfaz usando un NavigationView que nos permitirá más adelante navegar entre vistas y una lista (List) que formatee aquello que veamos en pantalla en forma de una lista de elementos apilados uno encima del otro. Usamos un List en lugar de un VStack dado que, entre otras cosas, nos permitirá manejar eficientemente listas que puedan crecer y tiene funcionalidades integradas muy interesantes como el scroll directo sobre la lista y la eliminación de elementos. Dentro de la lista definimos el componente Toggle() usado para filtrar los elementos y un ForEach que iterará sobre cada elemento de la lista de amigos. Si soloFavoritos es falso, renderizará todos los elementos de la lista a través de una instancia de VistaFila. Caso contrario, renderizará solo aquellos amigos marcados como favoritos. Sobre cada instancia VistaFila definimos un link tal que al presionar dicha instancia nos redirija a VistaDetalle y se renderice los detalles de aquel amigo. El título que tendrá nuestra vista actual será 'Amigos'.



Como se puede apreciar, para seguir con el desarrollo de la actividad fue necesario parametrizar VistaDetalle y, por tanto, las vistas que la componen. Para ello, en primer lugar, definiremos dos propiedades. Por un lado, al igual que antes, usamos el property wrapper @EnvironmentObject para trabajar más cómodamente sobre el modelo de datos. Por otro lado, se tiene amigoCurrent que define al amigo actual. De forma análoga, modificamos las respectivas vistas que la componen para parametrizarlas de forma que los datos que en ellas aparezcan sean los almacenados en la instancia de amigo con la cual se encuentra trabajando.

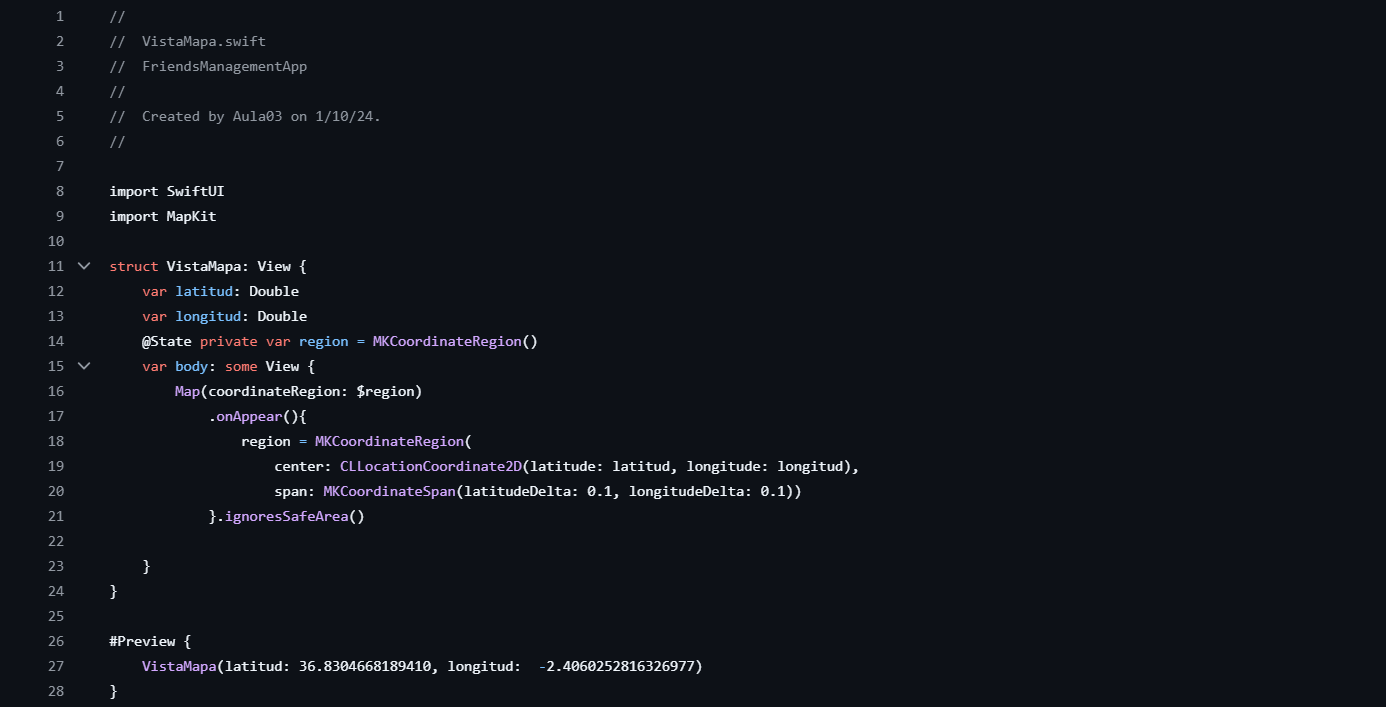
Algunas novedades que encontraremos son el uso de ScrollView en VistaDatos para hacerlo scrolleable junto al botón de favorito, la caja de texto para introducir alguna opinión acerca del amigo y la barra de progreso relativa a dicha caja de texto. De cara a trabajar con el valor original y no su copia, pues al trabajar con instancias de Amigo (tipo struct) enviamos copias, a partir del id de la copia encontramos el índice de la instancia dentro del modelo de datos a modo de trabajar directamente sobre dicho elemento en el modelo. Usaremos la función firstIndex.

Tras ello, en lo relativo al botón de favorito, será tan simple y directo como agregar un botón asociado al favorito del amigo dentro del modelo mediante un toggle(). Luego definimos la caja de texto como un TextEditor. Al igual que en VistaMapa, dicho componente necesita una variable de estado para manejar el contenido del editor, pero enfrentamos el mismo desafío: las variables de estado (@State) se inicializan antes que las propiedades regulares de la vista. Para resolver esto, seguimos la misma estrategia: en lugar de intentar inicializar la variable de estado con valores que aún no están disponibles, utilizamos el modificador onAppear() para establecer su valor cuando el componente se renderiza por primera vez. Esta aproximación garantiza que todas las propiedades necesarias estén correctamente inicializadas antes de asignar el valor inicial al editor de texto.

Hecho esto, la longitud del texto dentro del campo se limitará a 150 caracteres. Para ello, definimos el modificador onChange que observa todo cambio en la variable de estado previamente definida y recibe el nuevo valor como newValue. Luego, la longitud del nuevo valor la comparamos con la máxima permitida definida en una variable. Si su longitud es superior, entonces limitamos el texto a la longitud permitida y el texto obtenido se guarda en la variable de estado para que sea lo que aparezca en la caja de texto. Nótese que hacemos un casting explícito a String. Ello se debe a que prefix devuelve un Substring, no un String.

Finalmente, tenemos la barra de progreso implementada usando el componente ProgressView. Dicho componente recibe el valor actual como la longitud actual de la variable de estado de la cual antes hemos hablado y la longitud máxima permitida. Recordemos que cada cambio en la variable de estado renderiza nuevamente la pantalla. Así pues, cuando la longitud de la variable de estado antes mencionada es igual o superior a 100, se coloreará de rojo el texto que encapsula la barra de progreso.





Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

Con el fin de terminar lo relativo a esta sesión se muestra una fotografía de nuestro punto de entrada detallado con brevedad anteriormente:

Texto

Descripción generada automáticamente con confianza baja

Antes de concluir la sesión, es importante analizar una cuestión pendiente: la posibilidad de eliminar la variable amigoCurrent y, en su lugar, trabajar únicamente con la posición del array donde se encuentra el amigo que estamos editando, tal como hemos implementado en VistaDatos. Esta propuesta presenta tanto ventajas como desventajas que merecen ser analizadas.

Por un lado, trabajar con un índice de tipo entero en lugar de una instancia completa reduciría el consumo de memoria. Sin embargo, esta aproximación presenta dos inconvenientes significativos:

1. Mayor riesgo en términos de seguridad al trabajar directamente sobre el modelo de datos en componentes donde solo se requieren operaciones de lectura.
2. Mayor acoplamiento con la estructura de datos actual, lo que podría complicar futuras modificaciones. Por ejemplo, si se decidiera cambiar de un array a un diccionario, el uso de índices enteros dejaría de ser válido y requeriría una refactorización significativa del código.

Considerando estos factores, en nuestra implementación hemos optado por mantener la variable amigoCurrent. Esta decisión prioriza la seguridad y flexibilidad sobre la optimización de memoria.

Ejercicio 04: Comprueba que los cambios se quedan guardados en  
el modelo de datos. **Ejecutando la app el simulador, navega entre  
pantallas, cambia datos, filtra y, en todo momento, comprueba que  
todo funciona correctamente. Documenta todos los pasos que has  
seguido para comprobar la correcta funcionalidad de tu app.**  
Pista: ¿Evento .onDissapear() para guardar los cambios en el campo  
de texto? Cuidado a la hora de elegir el componente al que asociamos  
la gestión de eventos. Quizá no funcione bien si asociamos este evento  
con el editor de texto…quizá sea mejor con el stack de primer nivel.  
Y ya que estamos, puede ser que veas bien asociar los gestores de los  
eventos .onAppear() y .onDisappear() con el mismo componente.  
**Comenta tu decisión ¿?.**

Al final emplee dichos eventos sobre el mismo componente TextEditor pues fallos alguno no me dio y permite una mayor legibilidad del codigo

Mejoras futuras:  
¿Solo 3 amigos? ¿Y si quiero más? ¿Y si alguno alcanza un status por  
debajo de “no favorito” y quiero eliminarlo de mi lista? ¿Y si quiero  
ordenar la lista? ¿Podemos hacer que esta lista sea persistente? ¿Qué  
elijo, base de datos local o en servidor?

**¿Se hace?** Creo que se refiere a lo que se ve en las siguiente sesiones

# Sesion 04 - Xcode, Swift, SwiftUI e iOS: Gestión dinámica de listas y TabView

En las sesiones previas de desarrollo de nuestra aplicación GestionAmigos, hemos implementado funcionalidades fundamentales que incluyen la gestión de una lista estática de amigos, filtrado por favoritos, navegación detallada de registros, modificación de atributos en tiempo real, y gestión de opiniones con limitación de caracteres y feedback visual mediante una barra de progreso.

La cuarta sesión tiene como objetivo expandir significativamente la funcionalidad de la aplicación mediante la implementación de nuevas características clave:

1. Reorganización de la interfaz mediante un sistema de pestañas (TabView), incorporando nuevas vistas como VistaMain, VistaIdentificacion y VistaChat
2. Implementación de un sistema de registro de usuario en el dispositivo
3. Integración de una barra de búsqueda para facilitar la localización de amigos
4. Ampliación de la gestión de amigos, permitiendo:

* Inserción de nuevos contactos
* Reordenación de la lista
* Eliminación de registros existentes

1. Incorporación de una funcionalidad básica de chat simulado

Esta sesión se centra en mejorar la experiencia de usuario y ampliar las capacidades de gestión de datos, transformando nuestra aplicación en una herramienta más completa y funcional de gestión de contactos.

Con motivo empezar a tener muy presente el patrón de diseño base de SwiftUI conocido como MVVM (acrónimo del inglés Model - View - ViewModel) organizaremos los archivos de forma lógica haciendo uso de grupos. Por un lado, encontramos el grupo Model donde encontramos archivos que guardan la estructura de entidades. Por otro lado tenemos, el grupo ViewModel que guarda al modelo de datos con el cual estaremos trabajando a lo largo de las vistas. Nótese que un modelo de datos es un modo de conexión usada por la vistas para trabajar sobre las entidade (modelos). Dicha vista de modelo de datos puede inicializar el modelo (como en nuestro caso) así como proponer métodos y funciones para trabajar directamente sobre el modelo añadiendo una capa de aislamiento y protección al modelo. Finalmente tenemos los grupos de Vistas donde definimos las diversas vistas de nuestras aplicación. Recordad que en la vista es donde se realiza la maquetación visual de la aplicación. **(echar vistazo estructura de directoriosdel profesor y modificar la nuestra para hacerla como la suya)**.

En el desarrollo de aplicaciones SwiftUI, frecuentemente surge la cuestión de si es más conveniente implementar una subvista en un archivo separado o mantenerla en el mismo archivo. Esta decisión dependerá del caso específico y del criterio del ingeniero. Por ejemplo, podríamos cuestionar la necesidad de implementar la celda de ListaAmigos en un archivo separado como VistaFilaAmigos. Sin embargo, existen componentes que se utilizan con frecuencia, como las imágenes con borde redondo y sombra, donde resulta beneficioso crear un componente parametrizado mediante extensiones de Swift.

Las extensiones en Swift son un mecanismo que permite añadir funcionalidad a tipos existentes (clases, estructuras, enumeraciones o protocolos). En nuestro caso, podríamos crear una extensión de View que incluya un modificador personalizado para aplicar estilos comunes a las imágenes. Por ejemplo, podríamos definir un modificador circularImageStyle que aplique el recorte circular, el borde blanco y la sombra roja que utilizamos repetidamente en nuestra aplicación. Esta aproximación ofrece ventajas significativas: reduce la duplicación de código, mejora la mantenibilidad (los cambios se realizan en un único lugar), proporciona flexibilidad mediante parámetros personalizables y ayuda a organizar mejor el código. Además, el uso de extensiones hace que el código sea más limpio y siga el principio DRY (Don't Repeat Yourself), facilitando futuras modificaciones y mejoras en la aplicación.

**(Notar que no hemos renombrado casi ningún de los archivos que hemos usado. Nos pareció innecesario)**

La estructura VistaMain implementa la vista principal de nuestra aplicación utilizando un sistema de pestañas mediante TabView. Esta vista se inicializa con dos propiedades fundamentales: modeloDatos, marcada con @EnvironmentObject para acceder al modelo de datos compartido, y seleccion, una variable de estado (@State) inicializada a 0 que controlará qué pestaña está actualmente seleccionada.

El componente TabView recibe como parámetro selection un binding a la variable seleccion (indicado por $seleccion), estableciendo así una conexión bidireccional entre la interfaz y el estado. Es importante recordar que el binding es un concepto fundamental en SwiftUI que permite la conexión bidireccional entre una propiedad y una vista por medio de la variable de estado precedida del símbolo $, similar a como lo utilizamos anteriormente en componentes como Toggle(isOn: $soloFavoritos) para el filtrado de amigos o TextEditor(text: $textoOpinion) para la edición de texto. Esto significa que cuando el usuario toca una pestaña diferente, SwiftUI actualiza automáticamente el valor de seleccion para reflejar la pestaña seleccionada, y viceversa: si se modifica programáticamente el valor de seleccion, la interfaz actualizará la pestaña mostrada. Cada pestaña se identifica mediante un .tag() (0 para Identificación, 1 para Lista de Amigos, y 2 para Chat) que corresponde con los posibles valores de seleccion.

La vista incluye tres pestañas: VistaIdentificacion, ListaAmigos y VistaChat. Cada una se configura con su propio tabItem que define el ícono y texto mostrado en la barra de pestañas. Para ListaAmigos y VistaChat se inyecta explícitamente el modelo de datos mediante .environmentObject(modeloDatos), asegurando que estas vistas tengan acceso al modelo compartido. La vista previa se configura inicializando VistaMain con una nueva instancia del modelo de datos, permitiendo su visualización en el canvas de Xcode.

La estructura VistaIdentificacion implementa una vista que actúa como pantalla de identificación del usuario en nuestra aplicación. Para gestionar la persistencia del nombre de usuario, utiliza @AppStorage("userName"), una versión simplificada de UserDefaults en SwiftUI que funciona como una mini base de datos local, ideal para almacenar pequeñas cantidades de datos (hasta 512KB) que se cargan en memoria al iniciar la aplicación. Esta vista maneja tres propiedades fundamentales: userName marcada con @AppStorage para la persistencia del nombre en el dispositivo, currentUserName como variable de estado para manejar el nombre durante la edición, y cambiarNombre para controlar la visibilidad del campo de edición.

La interfaz se construye mediante un VStack que contiene una imagen de bienvenida estilizada con bordes redondeados, sombra y borde blanco. Debajo de ella, se muestra un HStack con el saludo al usuario y un botón con icono de teclado que, al pulsarlo, activa una animación que muestra u oculta el campo de edición del nombre mediante toggle(). Cuando cambiarNombre es verdadero, aparece un TextField que permite al usuario modificar su nombre, implementado dentro de un HStack centrado.

El flujo de datos se gestiona mediante dos modificadores clave: onAppear, que inicializa currentUserName con el valor almacenado en userName o 'desconocido/a' si no existe valor previo, y onDisappear, que guarda el nombre actual en userName solo si no está vacío. Esta implementación permite que la aplicación mantenga la persistencia del nombre del usuario entre sesiones, proporcionando una experiencia de usuario consistente y personalizada. Cuando el usuario abre la aplicación, se comprueba automáticamente si existe un usuario registrado, y siempre se le da la opción de cambiar su nombre, manteniendo así la flexibilidad en la identificación del usuario.

(Nota: quizás detallar sobre los modificadores .onAppear y .onDisappear en sesiones anteriores)

(Nota: checkear si dado esta sesión el modelo de datos era persistente o solo persistente [datos modificados se guardan] únicamente en la misma sesión)

Tras haber realizado VistaAutenticacion, pasemos a realizar VistaChat. Como era de suponer hemos realizado solo una maquetación no funcional de un chat dejando el modelo y estructura básica por si en el futuro deseamos implementar la funcionalidad. Para ello, primeramente, tuvimos que crear una nueva entidad llamada Mensaje e introducir un nuevo atributo en la entidad Amigo consistente en un array de instancias de la estructura Mensaje. Como vemos la nueva entidad Mensaje adopta el protocolo Hashable. La razón detrás de ello es que más adelante querremos iterar sobre el array de instancia de la estructura Mensaje para generar las correspondientes vistas. Para ello bien recordamos necesitaremos un forEach. Sin embargo, el forEach exige que los elementos del array sobre el cual itere puedan identificarse. Previamente no tuvimos ningún problema en iterar sobre elementos de un array de Amigos pues esta entidad adoptaba el protocolo Identifiable y nos permitía crear explícitamente un identificador único para cada elemento. Ahora bien, sería redundante crear un identificador único para cada mensaje. Entonces surge la idea de identificarlo implícitamente a través de sus propiedades. Por dicha razón la nueva entidad adopta el protocolo Hashable. El protocolo Hashable es una extensión del protocolo Equatable. Este ultimo protocolo permitía utilizar el operador de igualdad (==) para comparar dos elementos habiéndose definido manualmente un criterio de comparación. Ahora bien, Hashable al ser extensión de Equatable redefine dicho compartamiento al definir como cirtiero de comparación en el que todas sus propiedades sean iguales constituyendo cada elemento, así pues, un hash, es decir, una clave única por si solos. Como veremos más adelante en la implementación de VistaChat, cuando iteremos sobre la estructura de Mensaje con un forEach deberemos especificar explicitamete el id. En nuestro caso, como bien hemos dicho el hash, la clave o el id, como bien quieras llamarle, es el propio elemento, entonces usamos *id: \self*

COLOCAR FOTO

Hecho lo anterior también tuvimos que modificar el modelo de datos para añadir a nuestra lista de amigos una lista de mensajes estáticos para cada uno para realizar la implementación de la futura VistaChat

COLOCAR FOTO

Hablando propiamente de VistaChat, esta vista, como bien hemos venido haciendo con anterioridad, define el property wrapper @EnvironmentObject con el fin de establecer conexión con el modelo de datos. La vista se estructura mediante una List principal que organiza las conversaciones por amigos. La vista aprovecha la potencia de los protocolos Swift para gestionar eficientemente sus colecciones: mientras que los amigos se iteran directamente mediante ForEach gracias a su conformidad con Identifiable (que proporciona un id único), los mensajes requieren la especificación explícita id: \.self debido a su adopción del protocolo Hashable, que utiliza la combinación de sus propiedades (fecha, texto y estado de contestación) como identificador único. La vista se enriquece con elementos visuales informativos, incluyendo un encabezado personalizado (VistaHeader) para cada amigo, iconos direccionales que indican si los mensajes son enviados (flecha roja) o recibidos (flecha verde), y fechas formateadas mediante la función formatDate, que utiliza DateFormatter para presentar las marcas temporales en un formato localizado en español: .dateStyle = .short muestra la fecha en formato corto (dd/MM/yy), .timeStyle = .short muestra la hora en formato corto (HH:mm) y locale = Locale(identifier: "es") asegura que se use el formato español (día/mes/año). Un ejemplo del formato obtenido sería 15/11/24 13:45. La vista también implementa un pie de sección que muestra el conteo de mensajes con la pluralización correcta.

COLOCAR IMAGEN

Hecho lo anterior, el siguiente paso fue modificar nuestra vista ListaAmigos para incluir nuevas funcionalidades tales como una barra de búsqueda. La implementación de la barra de búsqueda en la vista ListaAmigos se ha realizado mediante la creación de una subvista llamada BusquedaView que se utiliza dentro de la misma. En vista ListaAmigos se define un estado local usando @State var query: String que almacenará el texto de búsqueda. BusquedaView se incluye dentro del VStack de ListaAmigos, justo antes de la List que muestra los amigos. La conexión entre ambas vistas se realiza pasando la variable query como un binding mediante BusquedaView(text: $query). El uso de @Binding permite crear una referencia bidireccional al valor de query, asegurando que cualquier cambio en el texto de búsqueda en la subvista se refleje directamente en la vista padre, manteniendo así la coherencia de los datos. El filtrado de los amigos se implementa dentro del ForEach que recorre el array de amigos, donde se añade una condición que comprueba si el query está vacío o si el nombre del amigo contiene el texto de búsqueda (ignorando mayúsculas y minúsculas) mediante la expresión query.isEmpty || amigoCurrent.nombre.lowercased().contains(query.lowercased()). De esta manera, la lista se actualiza dinámicamente mientras el usuario escribe en la barra de búsqueda, mostrando solo los amigos que coinciden con el criterio de búsqueda.

Quizás hablar acerca de que la vista que se instancia desde fuera es ListaAmigos y que por eso se dice es la vista principal del archivo, pero que realmente la vista principal del archivo seria aquella que se instancie. De ahí la importancia de tener nombres diferentes para vistas pues sino no se reconocerían y habrían fallos de conflictos porque XCode permite referenciar a una vista de cualquier archivo y esta detectaría de cuál archivo es.

FOTO

Tras esto último, implementamos la eliminación y ordenación de registros. Para ello, fue necesario modificar previamente VistaDatos. Para entender la razón, consideremos el siguiente escenario: desde ListaAmigos, accedemos a VistaDetalle del último amigo y modificamos su información en VistaDatos; al volver a ListaAmigos y eliminar el registro que acabábamos de modificar, se produce un error. Este error ocurre porque VistaDatos permanece en memoria manteniendo su antigua referencia al amigo eliminado, lo que provoca un conflicto.

La solución implementada consiste en trabajar en VistaDatos con copias locales en lugar de referencias directas al modelo de datos. Para ello, realizamos los siguientes cambios:

1. Trasladamos los modificadores .onAppear y .onDisappear del TextEditor al ScrollView
2. Definimos una variable de estado (@State) para el atributo favorito
3. Convertimos en variable de estado el índice que obtiene la posición del amigo por su ID

De este modo, al renderizarse la vista, inicializamos las variables de estado con los valores de la copia local del amigo, y al desaparecer la vista, actualizamos el modelo con estos valores. Esta implementación evita el error anterior, ya que aunque eliminemos un registro del modelo original, VistaDatos continúa trabajando con su copia local en memoria, manteniendo una referencia válida hasta que sea necesario actualizarla con nuevos datos. Nótese que Swift mantiene VistaDatos en memoria como una optimización de rendimiento, no arbitrariamente. Esto permite una navegación más fluida y rápida, ya que no tiene que reconstruir la vista desde cero cada vez que el usuario navega hacia atrás. Dicho esto, la implementación de la eliminación y reordenación de registros en ListaAmigos se consiguió aprovechando las características nativas que proporciona el componente List de SwiftUI. Cuando los elementos se renderizan dentro de un ForEach que está contenido en una List, automáticamente se habilita el comportamiento visual del deslizamiento hacia la izquierda y el botón rojo de eliminación, sin necesidad de programarlo manualmente. Para gestionar estos elementos se colocó un botón en la esquina superior izquierda de la barra de navegación (navigationBarItems con posición leading) que controla el modo de edición de la lista. Este botón alterna entre los estados "Editar" y "Cancelar" mediante la variable de estado enEdicion, que al activarse (true) muestra las tres líneas horizontales características del modo de edición junto a cada elemento de la lista. La activación del modo de edición se consigue mediante el modificador .environment que permite inyectar valores en el árbol de vistas, en este caso, inyecta el modo de edición (editMode) que puede estar activo o inactivo según el valor de enEdicion. Una vez en modo edición, es posible reordenar los elementos de la lista arrastrándolos mediante estas líneas, gracias al modificador .onMove que recibe los índices de origen y destino del elemento arrastrado. Además, independientemente del modo de edición, la lista implementa la funcionalidad de eliminación de elementos mediante el modificador .onDelete, que permite eliminar un elemento deslizándolo hacia la izquierda y recibe automáticamente el índice del elemento a eliminar. Ambas funcionalidades (.onMove y .onDelete) son capaces de identificar correctamente los elementos afectados gracias a que los elementos de la lista (Amigo) conforman el protocolo Identifiable, lo que permite a SwiftUI mantener un seguimiento preciso de la posición de cada elemento en la lista.

FOTO

Para implementar la funcionalidad de agregar amigos, se añadió un botón en la esquina superior derecha de ListaAmigos que, al ser presionado, activa la variable de estado @State var mostrarAddAmigo mediante toggle(). Esto, junto con el modificador .sheet, permite navegar a VistaAddAmigo, una vista que recibe mediante @Binding las variables de estado necesarias para crear un nuevo amigo (nombre, teléfono, email, imagenID) y una variable cancelar que indica si la operación se completó o se canceló. VistaAddAmigo utiliza @Environment(.presentationMode) para gestionar la navegación, permitiendo volver a ListaAmigos mediante modoPresentacion.wrappedValue.dismiss() tanto al guardar como al cancelar. Al guardar, cancelar permanece false, lo que activa el onDismiss del .sheet en ListaAmigos, creando un nuevo registro con la información proporcionada o, en su defecto, con valores predeterminados. Si se cancela, cancelar se establece a true y no se crea ningún registro. Cabe resaltar que para hacer funcionar el preview de VistaAddAmigos tuvimos que inicializar las variables @Binding. Para tener un código más limpio recurrimos al uso de valores constantes para su inicialización.

FOTO

**Ejercicio 02. Cuéntame cómo has implementado el componente de  
búsqueda y amplía el if teniendo en cuenta la variable query.  
Documenta (de manera informal) qué pruebas has hecho para  
comprobar que todo funciona correctamente.  
Página 19**

**Ejercicio 04. Como hemos ido construyendo la app paso a paso,  
haciendo cambios constantes, es muy probable que el código  
resultante sea muy “mejorable”. ¿Qué cambios propones para obtener  
un código de calidad?**

# Sesión 05 - Xcode, Swift, SwiftUI e iOS: Persistencia de datos - CoreData

En esta sesión final de nuestro recorrido por Swift, aplicaremos los conocimientos adquiridos para desarrollar una aplicación de gestión de personas y mascotas. La aplicación permitirá gestionar una lista dinámica y permanente  
de personas y de mascotas, teniendo en cuenta que 1 persona puede  
tener muchas mascotas, mientras que una mascota solo va a pertenecer  
a una única persona (la titular del registro). Este proyecto nos permitirá integrar diversos conceptos aprendidos, con especial énfasis en la persistencia de datos.

En el desarrollo de aplicaciones iOS, la persistencia de datos es un aspecto fundamental que requiere soluciones adaptadas a diferentes necesidades. Hasta ahora, hemos utilizado el property wrapper @AppStorage como herramienta básica para persistir datos simples, como nombres de usuarios. Sin embargo, sus limitaciones en términos de capacidad de almacenamiento y flexibilidad hacen necesario considerar alternativas más robustas para aplicaciones que manejan datos complejos.

En este contexto, CoreData emerge como el framework principal de Apple para la gestión avanzada de persistencia de datos. Este framework proporciona una capa de abstracción sobre el sistema de gestión de bases de datos relacionales, ofreciendo las siguientes características clave:

1. Arquitectura de Persistencia Robusta

* Abstrae las operaciones de bajo nivel en la base de datos
* Facilita la creación y gestión de modelos de datos estructurados
* Permite establecer relaciones entre entidades con generación automática de clases

1. Gestión Avanzada de Datos

* Permite el acceso a datos en modo offline
* Implementa sistema de caché temporal
* Ofrece control de versiones de operaciones
* Soporta funcionalidades de deshacer/rehacer cambios

1. Optimización de Recursos

* Ejecuta operaciones en segundo plano
* Gestiona la memoria eficientemente
* Reduce la latencia mediante almacenamiento en caché
* Minimiza accesos al servidor con almacenamiento local

CoreData representa así una solución integral para aplicaciones que requieren una gestión sofisticada de datos, superando las limitaciones de @AppStorage al proporcionar mayor capacidad de almacenamiento, estructuras de datos complejas, relaciones entre entidades y capacidades avanzadas de consulta y optimización.

**VERIFICAR SI SE TIENE LA MISMA ESTRUCTURA DE DIRECTORIOS**

Dicho esto, desde XCode creamos el proyecto como proyecto *single-view*, tal como hemos venido haciendo:  
  
FOTO

Organizando los archivos llegamos al siguiente resultado:  
  
FOTO

Hecho esto, el siguiente paso ha sido crear nuestro modelo de datos. El proceso seguido ha sido el siguiente:  
  
FOTO

NOTA: CORE DATA NOS LIBERA DE ESPECIFICAR DETALLES COMO CUAL SERA EL ATRIBUTO CLAVE O EL IDENTIFICADOR UNICO

Tras ello creamos el gestor de Core Data. Se trata de un archivo que implemente la lógica para comunicarnos con el modelo de datos antes creado. Tiene la siguiente estructura:  
  
FOTO

Como vemos el archivo define una clase llamada CoreDataManager que implementa el patrón Singleton para gestionar las operaciones de CoreData en la aplicación de gestión de mascotas. El patrón Singleton es un patrón de diseño que asegura que una clase tenga una única instancia en toda la aplicación y proporciona un punto de acceso global a ella, optimizando el consumo de memoria al mantener una única instancia compartida durante toda la ejecución de la aplicación y garantizando la consistencia de datos al proporcionar un único punto de acceso centralizado. Esta clase se encarga de inicializar y mantener la infraestructura necesaria para la persistencia de datos. Como habíamos anticipado el manager se inicializa creando una instancia única (singleton) mediante la propiedad estática 'instance', y establece dos componentes esenciales: el contenedor persistente (NSPersistentContainer) que se vincula con el modelo de datos "GestionMascotasModel", y el contexto de gestión (NSManagedObjectContext) que facilita las operaciones con los datos. El contexto será la variable utilizada para trabajar sobre el modelo de dato, actuando como una capa de abstracción que sirve de barrera de protección al acceso de los datos. Durante la inicialización, se carga el almacén persistente mediante loadPersistentStores, implementando un manejo de errores que informa si la carga fue exitosa o si ocurrió algún problema. Además, la clase proporciona un método 'save()' que permite guardar los cambios realizados en el contexto, asegurando la persistencia de los datos. Notifica mediante mensajes en consola si la operación fue exitosa o si se produjo algún error durante el proceso de guardado.

FOTO  
  
Una vez hemos hecho el paso anterior, el siguiente paso fue crear un intermediario entre el modelo de datos de CoreData y las vistas de la interfaz de usuario. Así pues nace la clase que hemos llamado ViewModel. La clase utiliza el gestor de CoreData previamente creado a través del singleton CoreDataManager.instance. A modo de notificar automáticamente los cambios en sus propiedades a las vistas adopta el protocolo ObservableObject. Mantiene dos arreglos publicados (@Published) para personas y mascotas, que se actualizan automáticamente en la interfaz de usuario cuando cambian. Una vez ha sido establecido conexión al modelo de datos y hemos creado las variables que estarán siendo accedidas desde las vistas de la interfaz de usuario para su futuro uso en ellas, el siguiente paso fue definir operaciones sobre el modelo de datos basándonos en las variables definidas. Primeramente tenemos el inicializador que llama al método que carga los datos del modelo de datos en las variables definidas. La carga de datos consiste en:

* Limpiar las variables
* Crear el resquest
* Por medio del *contexto* hacemos *fetch* usando el request antes definido al modelo de datos
* La respuesta es ordenada ascendentemente. Utilizamos el *desenvoltorio* (operador !) para acceder al valor de los atributos marcados como opcionales por defecto por Core Data

De igual modo, vemos el método guardarDatos(). La primera instrucción guarda los datos del contexto en el contenedor haciendolos persistentes empleando la función save() del modelo de datos. Luego vuelve a cargar los datos para mantener actualizado el programa.

Finalmente vemos las funciones para agregar y eliminar tanto personas como mascotas. Por un lado tenemos al método utilizado para agregar una persona. El método obtiene del contexto la estructura de la entidad, guardándolo en una variable. Dicha variable guarda relación con el contexto e inicializamos sus. Como bien recordamos hemos definido el tipo de una imagen como un *data binary*. Así pues, trabajaremos con instancias de UImage a modo de obtener una representación, por ejemplo, formato png, que nos permita trabajar con dicho tipo de datos. Finalmente guardamos el trabajo realizado. El proceso es el mismo al crear una mascota salvo que, además, hemos de establecer la relación con una persona. Con respecto al eliminar persona mencionamos que recibe un listado de índices. Itero sobre dicho listado y elimino las personas de acuerdo a su índice. Luego guardamos el trabajo hecho. Recordemos el listado de personas de la sesión anterior. Por último tenemos el eliminar una mascota. Este recibe como parámetro una mascota, la elimina y guarda el trabajo realizado.

FOTO

Hecho lo anterior, el siguiente paso fue ir construir la principal nuestra aplicación: ListaPersonasView. Dicha vista define un NavigationView con el fin de utilizar modificadores tales como el navigationBarItems y el navigationTitle. De forma análoga a sesiones anteriores, el primer modificador sitúa sobre el lateral superior derecho de la pantalla un botón de icono un signo + rodeado de un circulo que al ser presionado modifica el valor de la variable de estado mostrarAddPersona mediante un toggle() y permite agregar una nueva persona. El segundo modificador renderiza el titulo de la vista. Dentrodel NavigationView definimos VStack en el que dentro de el definimos un condicional y un List(). Hablando del List() en primera instancia, define un bucle ForEach que itera sobre las personas del modelo accedidas a través de @EnvironmentObject var vm: ViewModel. Para cada persona definimos un Section(). Un Section() es un elemento de SwiftUI que permite agrupar elementos comunes. Sobre el definimos un header mediante HeaderView() y los elementos relativos a tal sección se define como FilaView. Sobre cada persona permitimos su eliminación por medio de la funcionalidad integrado del List(). Cuando se realiza se llama al método de eliminación del ViewModel. No pasamos ningún identificador dado que el ForEach ya lo interpreta y lo envia implícitamente. Esto es una característica conveniente de Swift conocida como "method reference" o referencia a método. Cuando los parámetros del closure (en este caso, el handler de onDelete) coinciden exactamente con los parámetros del método al que se quiere llamar, podemos pasar directamente la referencia al método sin necesidad de escribir explícitamente los parámetros. Fuera del List() se encontraba el condicional. Este se activa al ser presionado el botón de agregar mencionado antes. Se renderiza VistaAddView que permite agregar una persona. Esta se renderiza como una fila sobre la vista.

FOTO

Respecto al HeaderView() siempre que el usuario tenga una foto se renderizara como una fila mediante un HStack. Dicha vista tiene un botón cuya imagen es la fotografía del usuario. Nótese usamos el *desevoltorio* para trabajar con los supuestos valores opcionales, pero que en realidad no lo son. Si el botón se presiona la variable de estado mostrarAddPersona cambia de valor y se muestra la vista AddMascotaView. Usamos el mismo esquema seguido para añadir la funcionalidad de agregar un amigo en la sesión anterior. La vista AddMascotaView no tiene mayor misterior. Dicha vista define un VStack sobre el cual sitúa un TextField para introducir el nombre de la mascota, HStack que define un texto para la edad al que a lado tiene un Slider. Como era de esperar dicho Slider, componente que proporciona SwiftUI, necesita recibir una variable de estado relativa a la edad para poder trabajar e ir actualizando la vista. En nuestro caso, @State var edad: Double = 0. Después definimos un componente Picker de SwiftUI para determinar si la mascota es perro o gato. Al igual que antes necesitaremos otra variable de estado pues buscamos actualizar la interfaz ante algún cambio. La variable definida es @State var tipoMascota: String = "Perro". Cuando se selecciona una de las opciones se se actualiza dicha variable al valor del tag, volviéndose a renderizar la vista, reflejando el cambio realizado. Finalmente tenemos los respectivos botones que permitan agregar la mascota o no. Para agregar la mascota tenemos el botón del pulgar hacia arriba que al ser presionado llama al método correspondiente del ViewModel para guardar los cambios, luego pone mostrarAddMascota a falso. Por otro lado, si se presiona el botón del pulgar hacia abajo significa que se ha cancelado la operación y simplemente cambiamos el valor de mostrarAddMascota a falso. Nótese que esta ultima variable es un @Binding enlazada a la referencia original en la vista ListaPersonasView que permite cerrar VistaAddMascota. De vuelta a HeaderView en ListaPersonasView mostramos el nombre del usuario al costado derecho de la imagen  
  
FOTO

Por último tenemos FilaView que mostrara las distintas mascota de una persona. Esta vista mostrara su contenido solo si la persona tiene mascotas, es decir, si se encuentra relación entre la persona y alguna mascota. Para ello se define la línea *if let mascotas = persona.mascotasRelation?.allObjects as? [MascotaEntity]*. Dicha línea accede a la relación de mascotas que es opcional (*persona.mascotasRelation?),* obtiene todos los objetos de la relación como un NSSet *(.allObjects) e* intenta convertir (cast) la colección a un array de MascotaEntity (*as? [MascotaEntity*). El resultado se guarda en la variable mascotas definida sobre el mismo condicional. Si la lista no es vacia, entonces es true y pasamos dentro del condicional. Dentro del condicional definimos un VStack que define un Text que muestra el numero de mascotas de la persona. Para ello accdemos al count() de la lista en mascotas. Luego iteramos sobre dicha lista. Dentro de un HStack definimos un Image que muestra la imagen de la mascota. Usamos el *desenvoltorio*. Luego mostramos el nombre y edad de la mascota. Finalmente, agregando un espacio entre medio, agregamos un icono del signo – rodeado por un circulo que al ser presionado accede al método de eliminación de una mascota del ViewModel

FOTO

FOTO

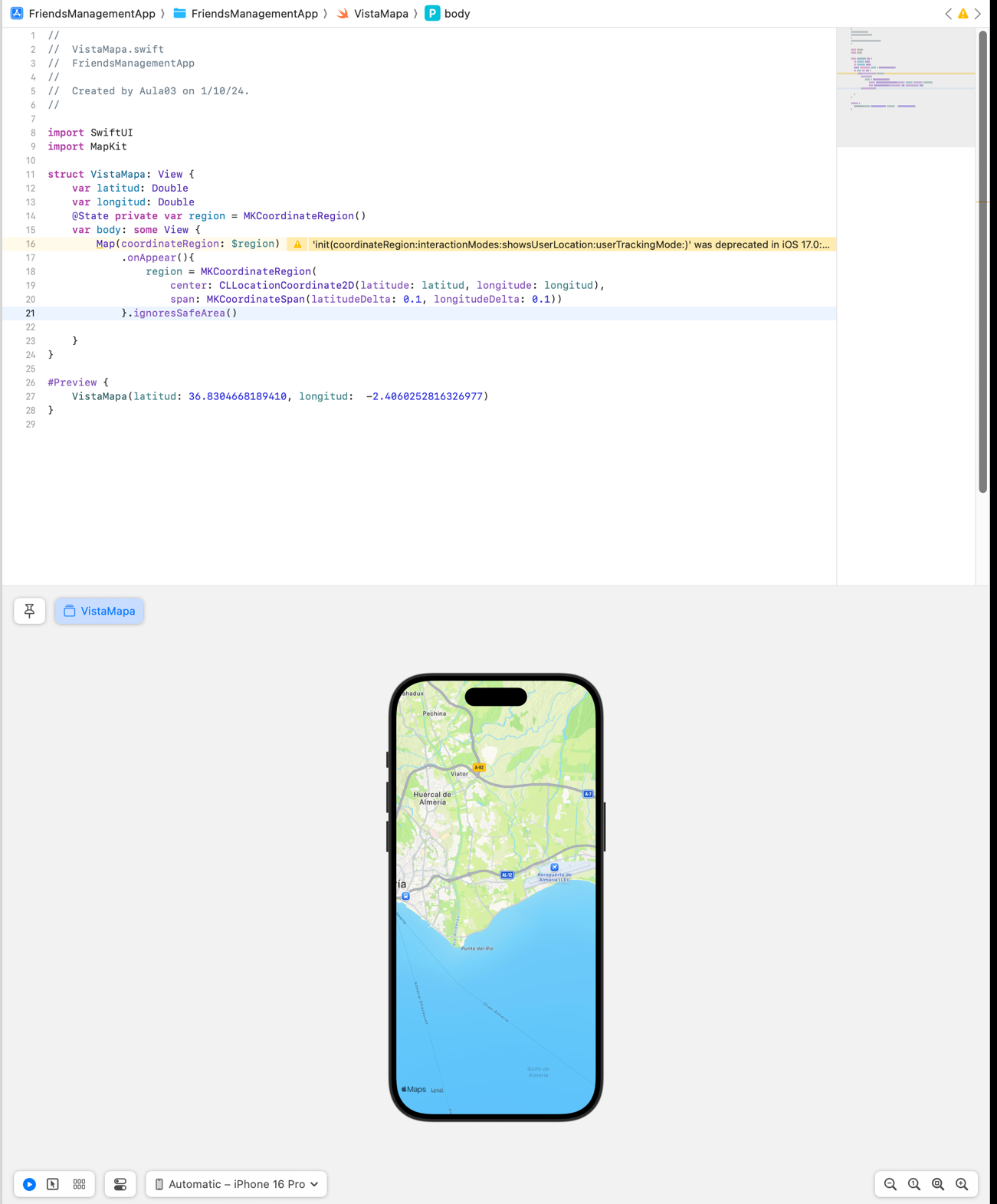
Habiendo definido el modelo de datos, la vista al modelo y las vistas, el último paso sería definir un punto de entrada a la aplicación conforme está sea compilada y ejecutada. El código del punto de entrada sería:  
  
FOTO

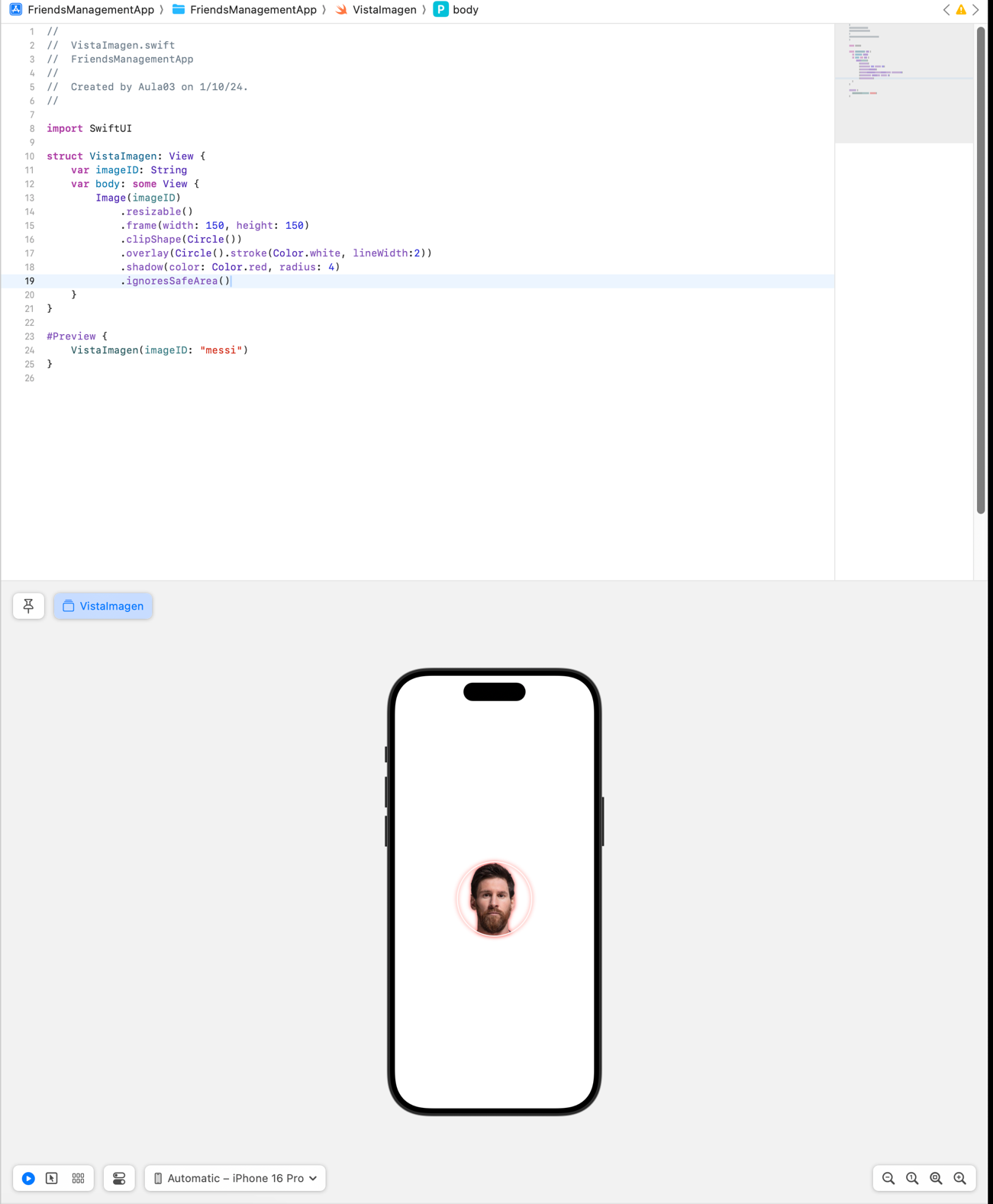
Al igual que en sesiones anteriores definíamos una objeto de estado para hacer seguimiento sobre el ViewModel. Tras ello, definimos la vistade entrada y le pasamos dicha vista de modelo como variable global que estará usando ella y las vistas que de ella luego estemos usando, mejorando la legibilidad del código.

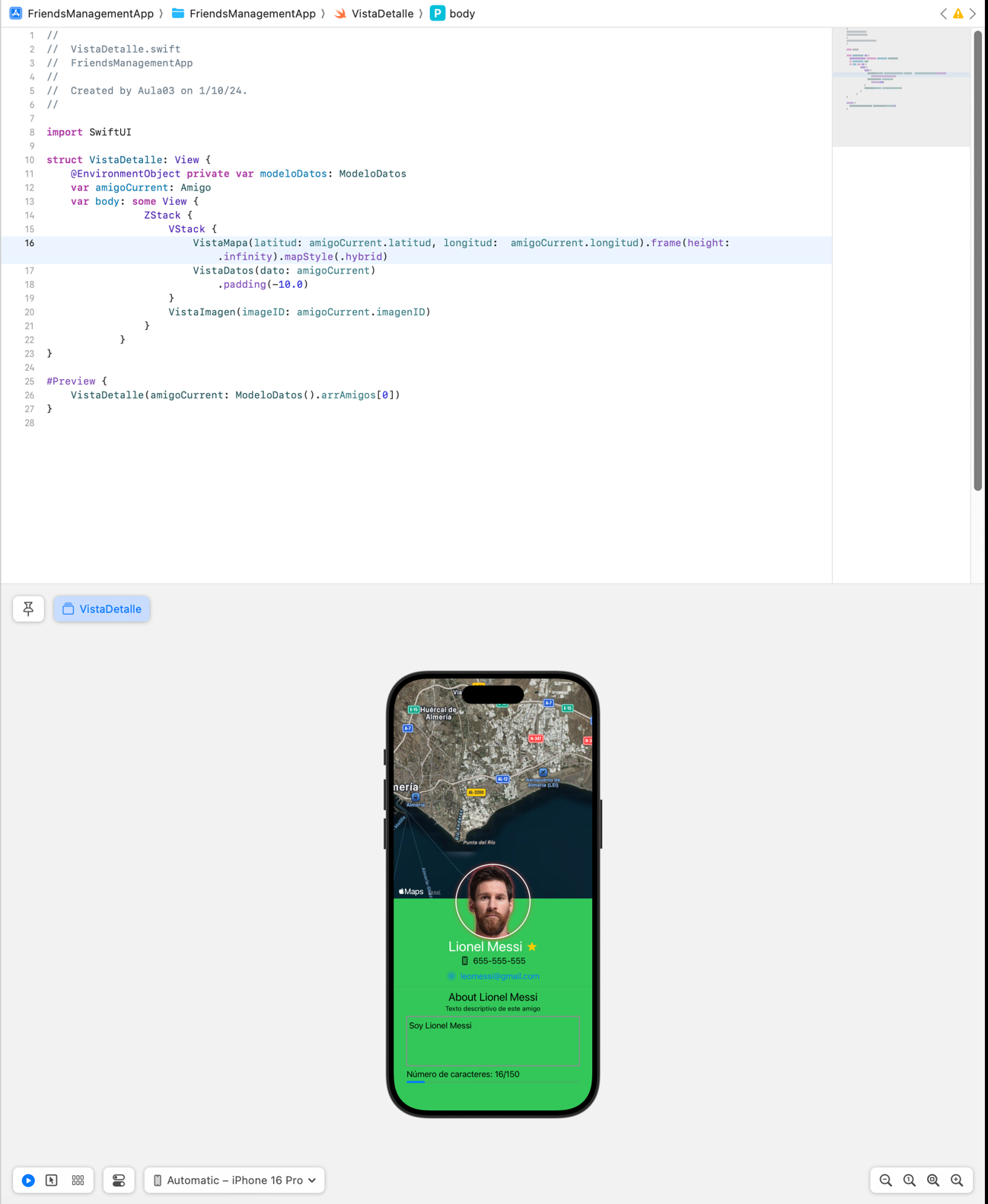
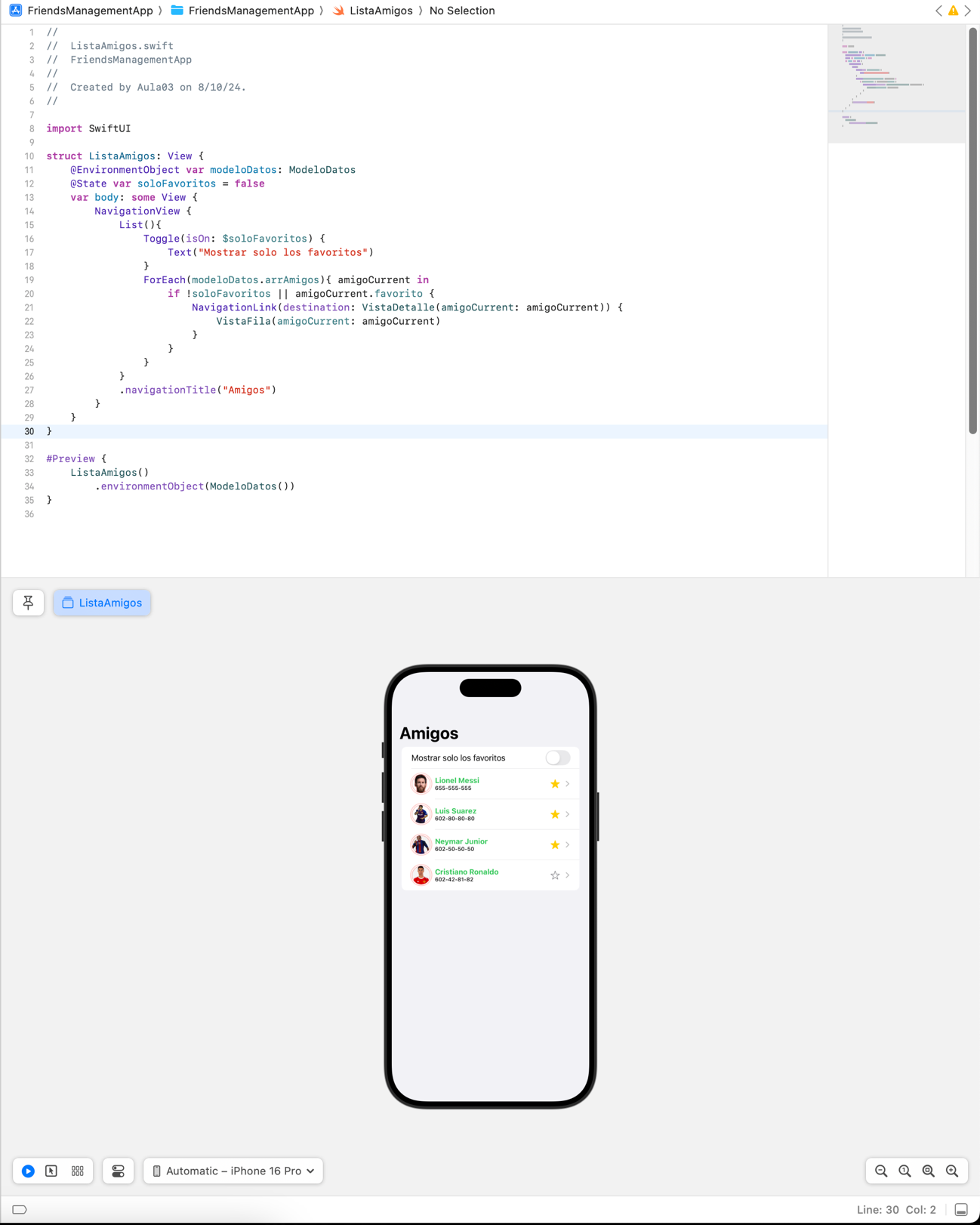
Cabe resaltar que no hemos trabajo usando ninguna preview. Sin embargo, en caso de haberla usado deberíamos de haber inicializado algunos valores para ir probando por medio del método addPersona comentado ante, notándose que al haber agregado, en la siguiente ejecución, dichos líneas deberían borrarse nuevamente pues los datos ya han persistido y mantener dichas líneas volvería a escribir los mismos datos otra vez (sin sobreescribir, sino agregándolos al modelo otra vez como datos nuevos e independientes a los previos)

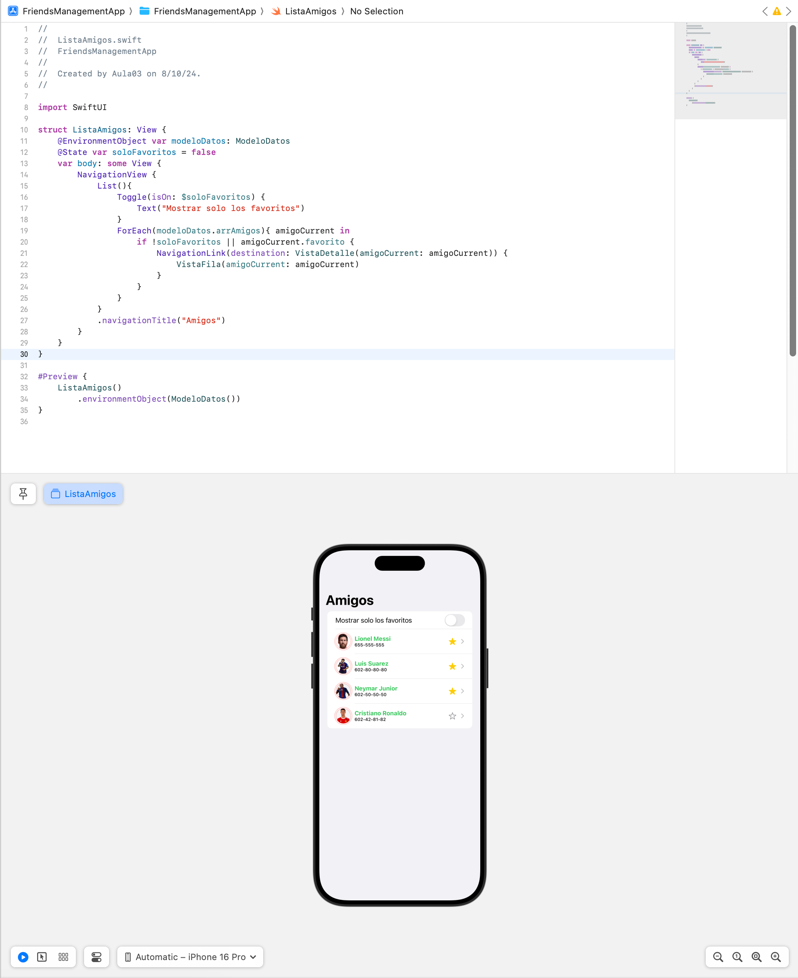
**Captura de pantalla de sesión 02**

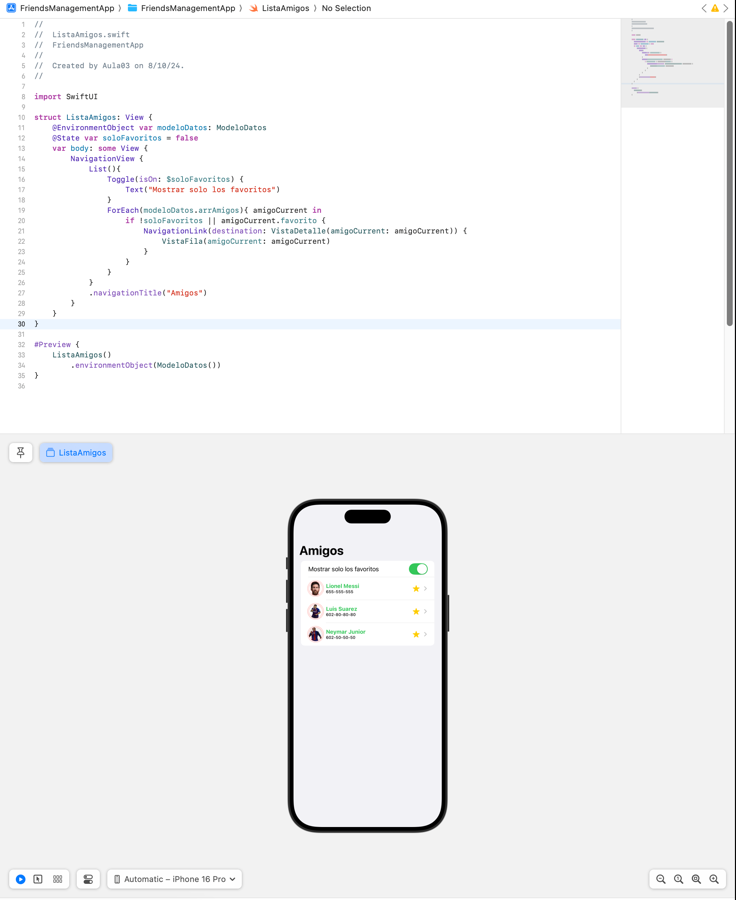
****

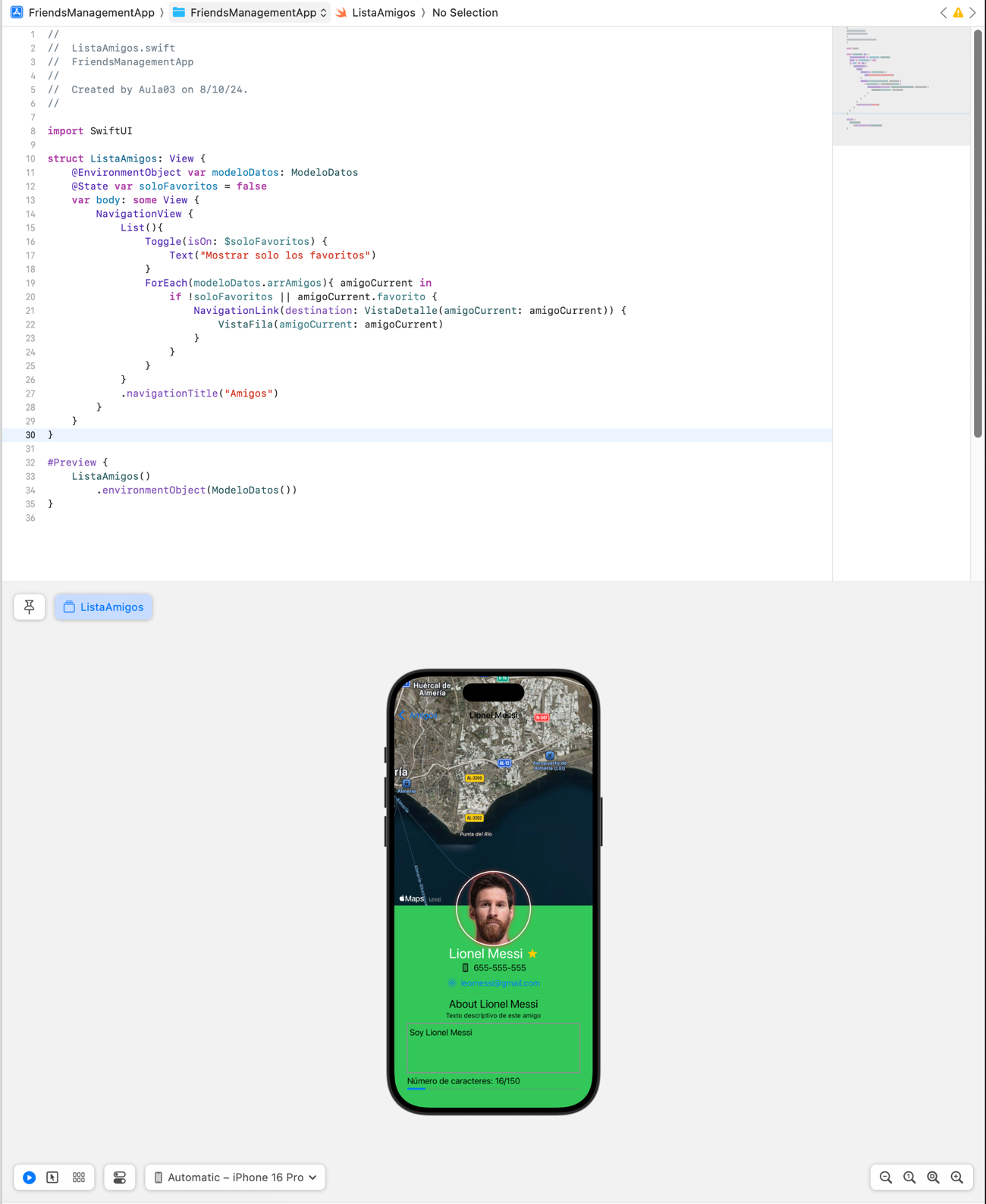
****

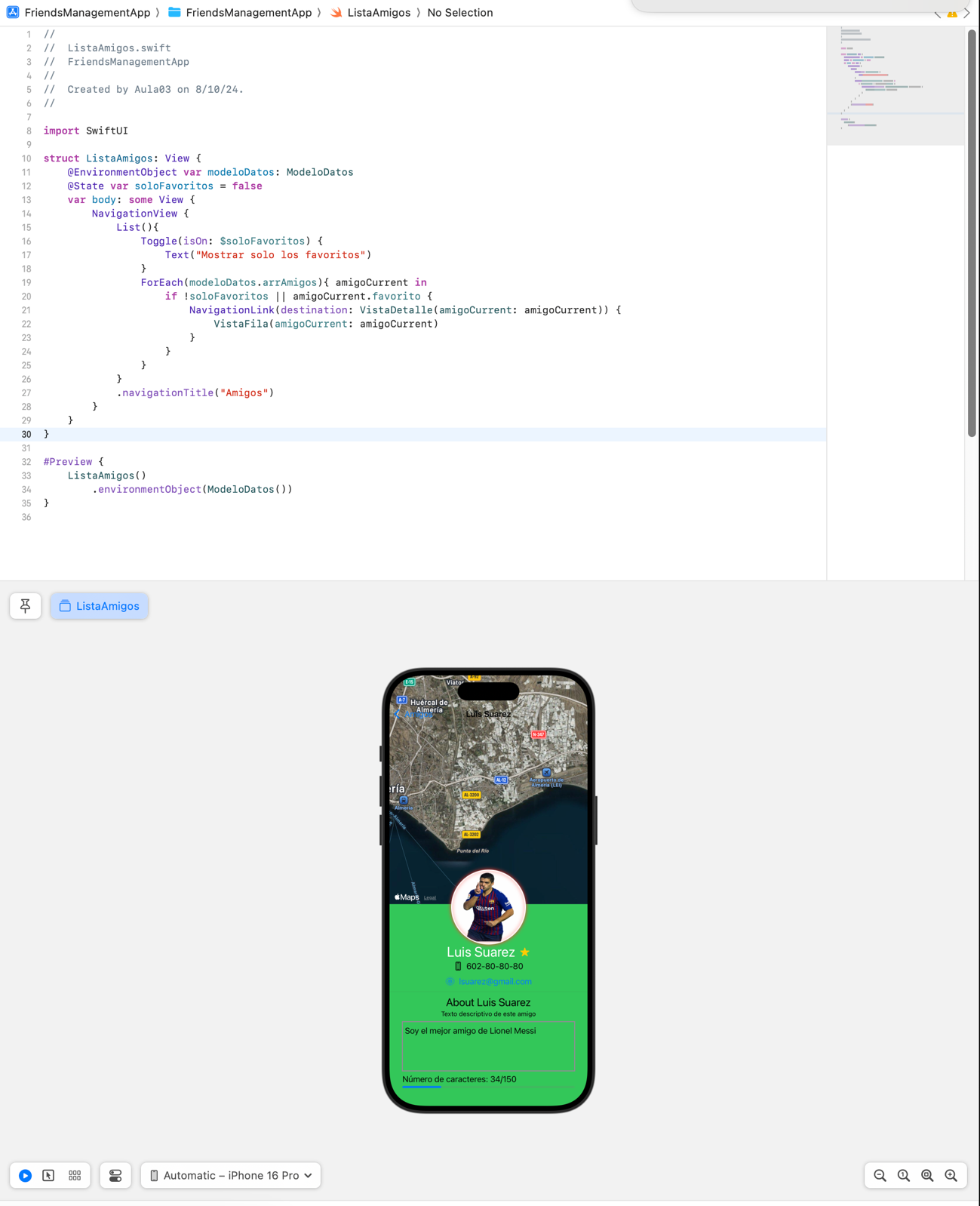
****

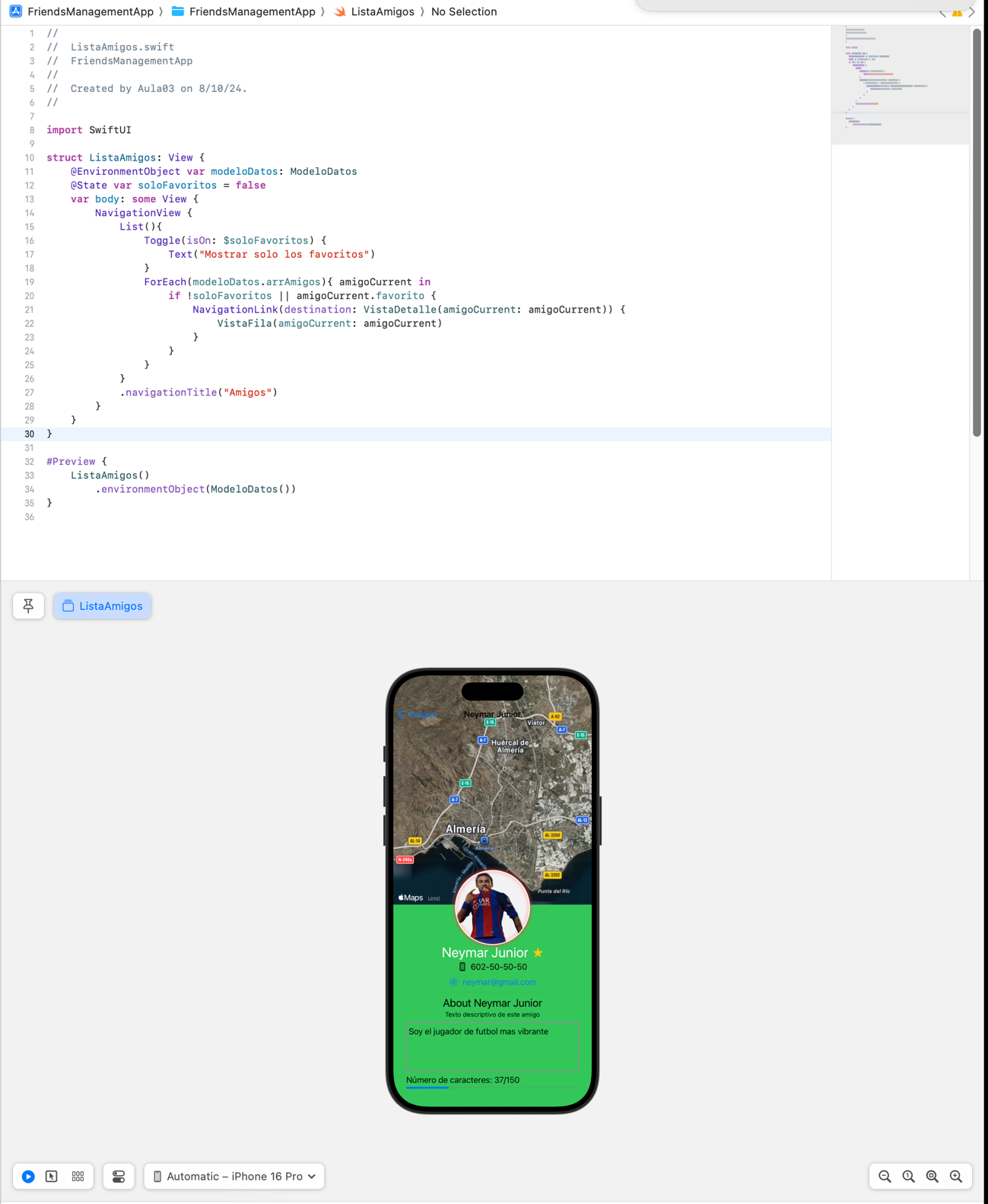
****

****

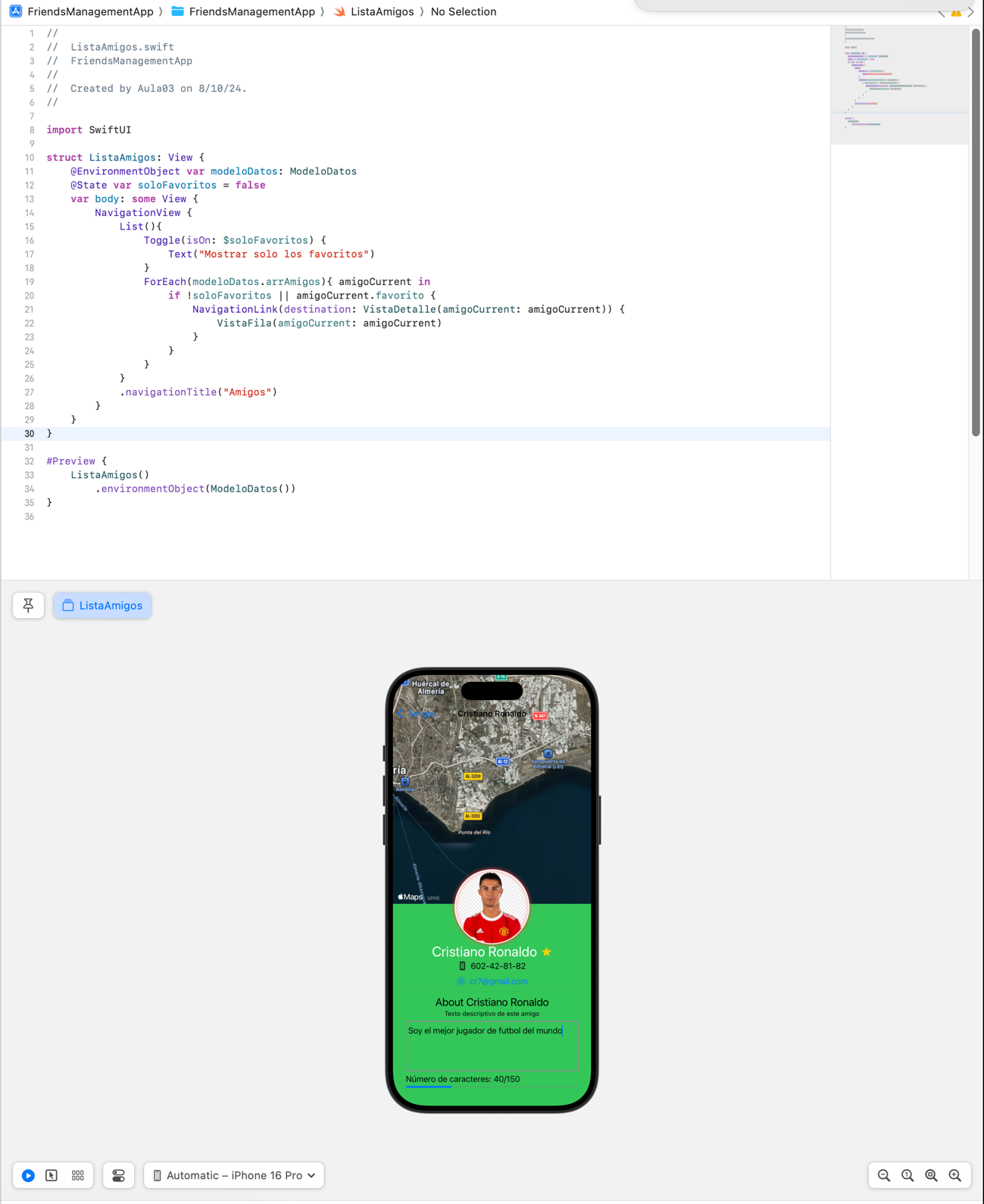
****

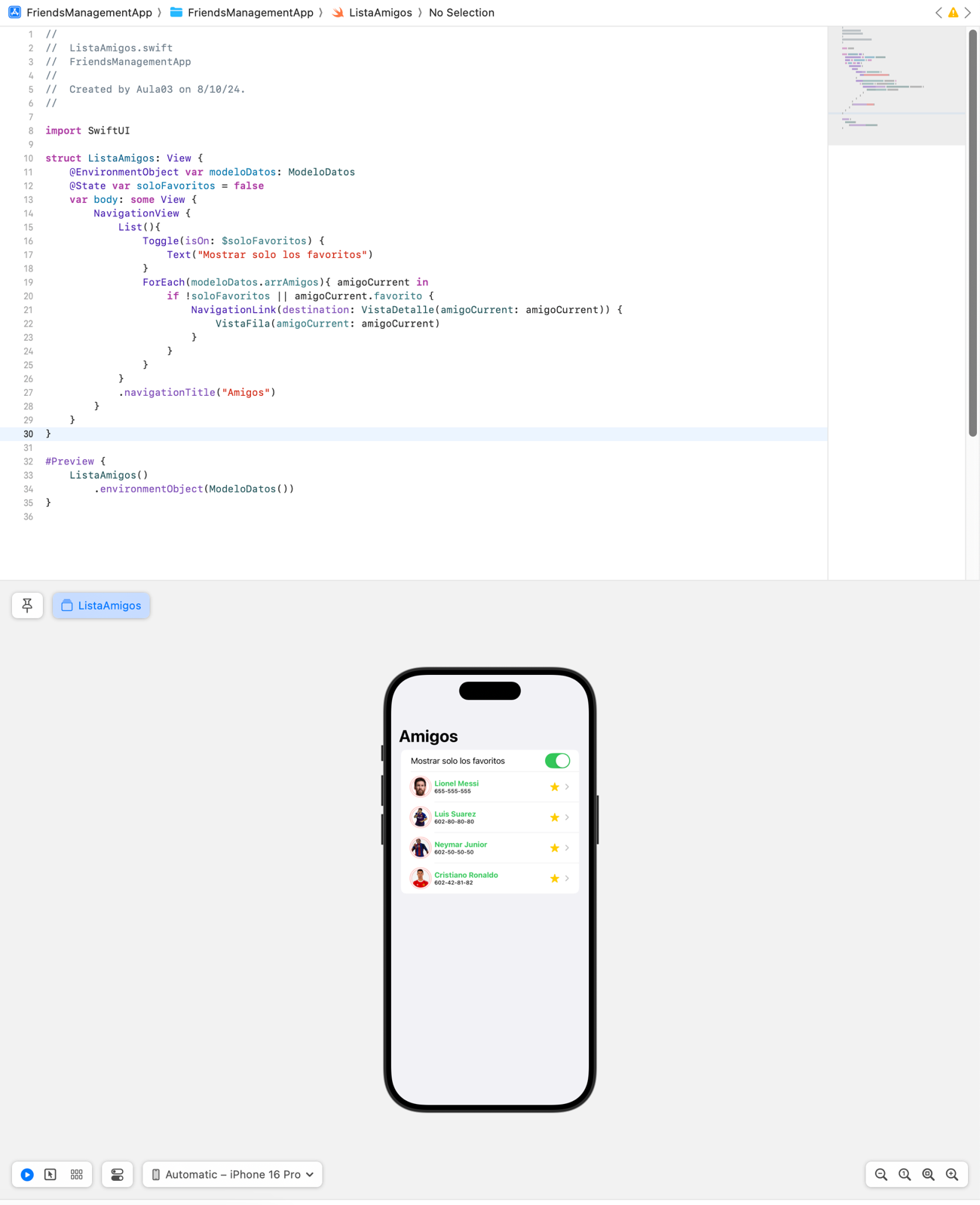
****

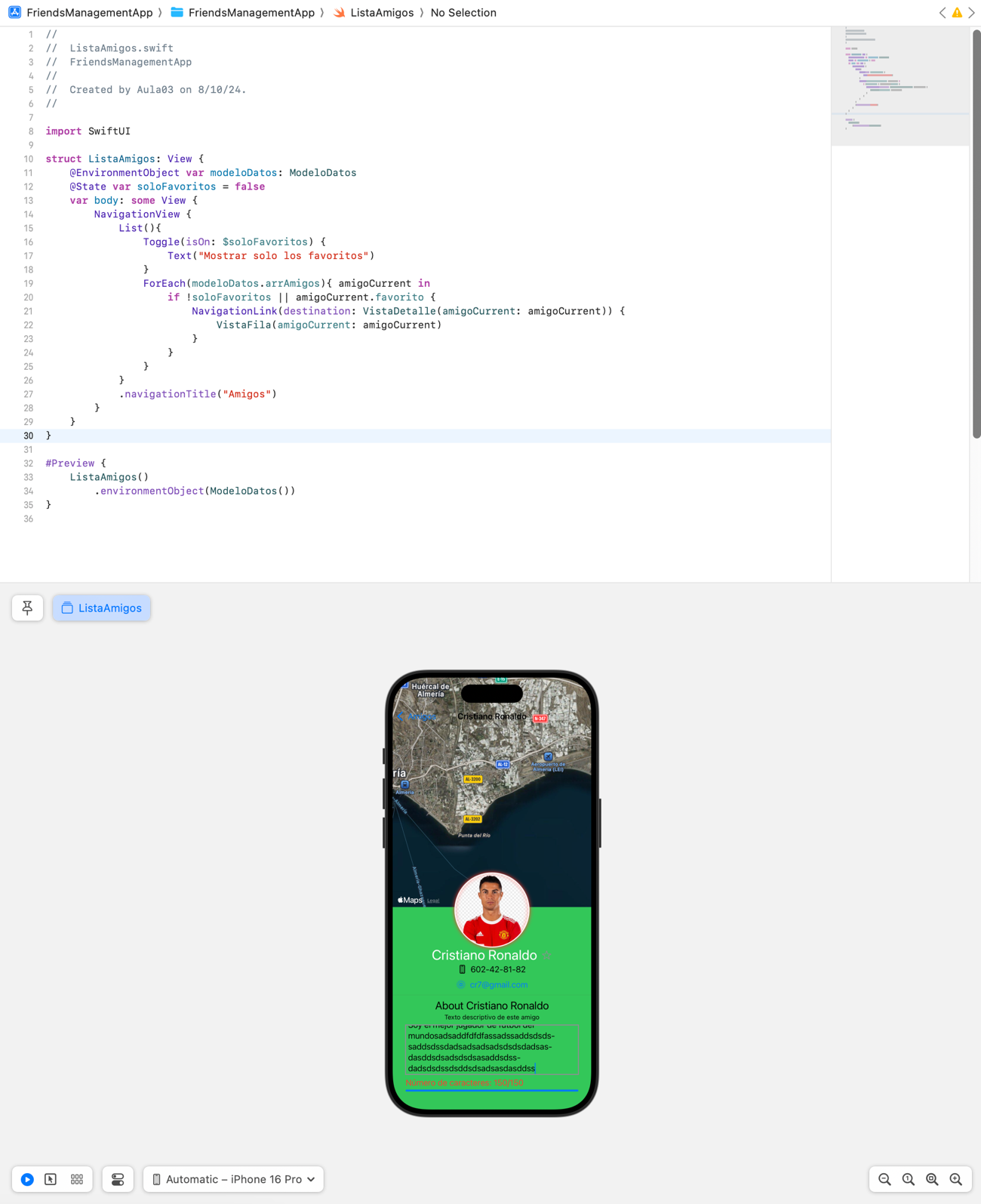
****

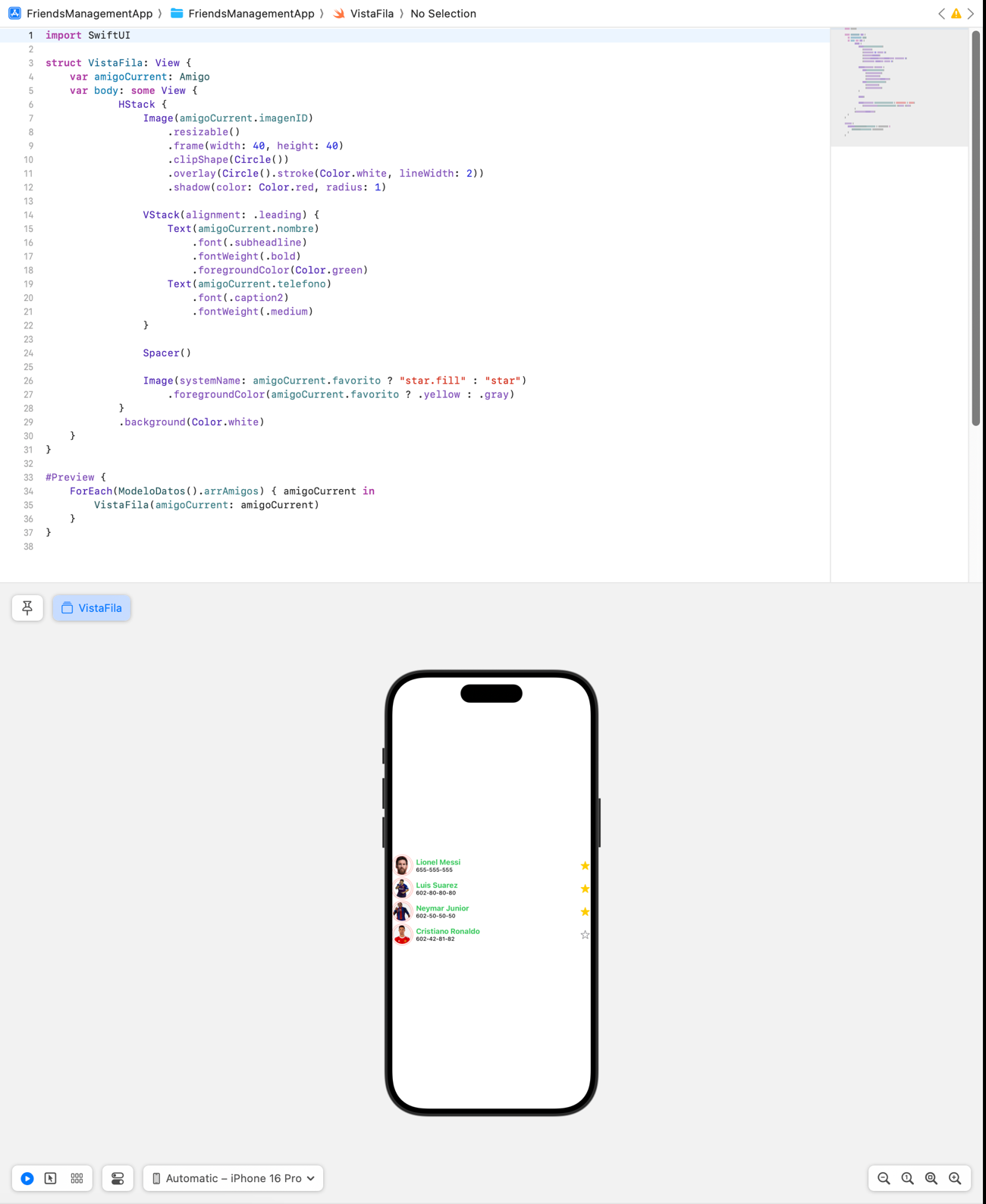
****

****

****

****

****

****