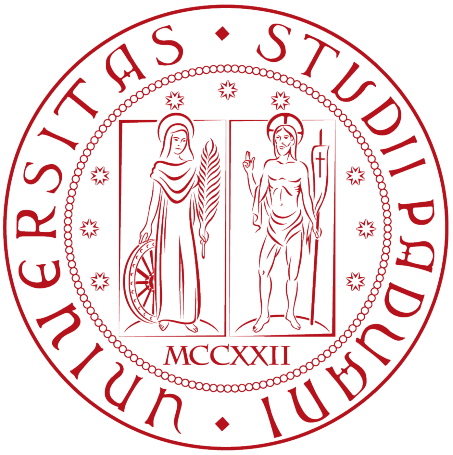
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Sfida e benessere digitale: l'evoluzione del fitness attraverso Apple TV

Tesi di laurea

Relatore

Prof.ssa Ombretta Gaggi

Laureando

Niccolò Fasolo 2000551

Anno Accademico 2022-2023

Here's to the crazy ones. The misfits. The rebels. The troublemakers. The round pegs in the square holes. The ones who see things differently. They're not fond of rules. And they have no respect for the status quo. You can quote them, disagree with them, glorify or vilify them. About the only thing you can't do is ignore them. Because they change things. They push the human race forward. And while some may see them as the crazy ones, we see genius. Because the people who are crazy enough to think they can change the world, are the ones who do.

— Apple Inc., spot pubblicitario "Think Different" (1997)

Sommario

La presente tesi descrive l'esperienza di stage di trecentoventi ore del laureando Niccolò Fasolo presso Rawfish s.r.l., durante la quale è stato affrontato lo sviluppo di una controparte per Apple TV di una piattaforma interna dedicata alla visualizzazione di contenuti audio e video. Lo sviluppo è stato realizzato utilizzando il linguaggio Swift e il framework SwiftUI. Tale framework permette di fornire un'esperienza interattiva e coinvolgente agli utenti, utilizzando componenti visivi nativi per la realizzazione dell'interfaccia utente dell'applicazione.
In seguito alla realizzazione della controparte per Apple TV, l'obiettivo è stato esteso all'analisi e all'implementazione di un'integrazione con Apple Watch e HealthKit. Questa integrazione ha consentito di visualizzare sullo schermo di Apple TV i dati biometrici dell'utente, come il battito cardiaco e le calorie consumate, oltre che a controllare la riproduzione del contenuto audio o video tramite lo stesso Apple Watch. Tale funzionalità si è rivelata particolarmente rilevante nel contesto di una possibile evoluzione della piattaforma verso un ambito legato al fitness e al benessere, dove risulta possibile tenere traccia dei propri dati biometrici durante la visualizzazione di un video workout, favorendo l'evoluzione dell'allenamento a casa.

"I want to put a ding in the universe"

— Steve Jobs

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine alla Prof.ssa Ombretta Gaggi, relatrice della mia tesi, per la sua guida preziosa e il suo supporto costante durante la stesura del lavoro.

Desidero inoltre ringraziare la mia famiglia per il sostegno, la pazienza infinita e il costante incoraggiamento che mi hanno dimostrato in ogni fase del mio percorso di studi.

Un caloroso ringraziamento va anche a tutti i miei amici che sono stati al mio fianco durante questi anni di studio. Il vostro sostegno morale e le vostre risate hanno reso questa esperienza indimenticabile.

Padova, Settembre 2023 Niccolò Fasolo

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei codici

Capitolo 1

Introduzione

In questo capitolo verranno introdotti gli aspetti principali relativi all'attività di stage. In particolare, verranno presentati l'azienda ospitante, il contesto in cui si è svolto il progetto e gli obiettivi che si sono prefissati.

1.1 Lo stage

Lo stage si è svolto presso l'azienda Rawfish s.r.l. di Vicenza, in via Carlo Mollino 90. Rawfish nasce nel 2012 come *software house* specializzata nello sviluppo di applicazioni mobile native, affiancando negli anni successivi lo sviluppo di *frontend* web e *backend* custom e un'area di UI/UX[G]. Attualmente vanta un portfolio progetti e clienti che poche altre agenzie italiane possono pareggiare.
Negli ultimi tempi Rawfish sta sviluppando una sua piattaforma *OTT* (*Over The Top*) interna per la distribuzione e riproduzione di contenuti audio e video in tutti i dispositivi di uso comune (mobile, browser, smart tv). In quest'ottica, lo stage ha previsto il mio inserimento all'interno del team iOS[G] con lo scopo di implementare uno dei *touch point* più importanti della piattaforma, ossia l'applicazione per dispositivi Apple TV[G].
Lo scopo è stato quello di apprendere le conoscenze idonee allo sviluppo di un applicazione su ecosistema *iOS* e tvOS[G] utilizzando il linguaggio nativo Swift[G] ed il *framework* SwiftUI[G]. Sono stato guidato dal tutor aziendale per imparare le *best practice* nel settore in termini tecnici e di collaborazione con il resto del team.
Dopo lo sviluppo dell'applicativo di base che permette login, scelta e visualizzazione del contenuto in *streaming*, l'attenzione si è spostata sull'analisi (e implementazione) di un'integrazione con HealthKit[G] e con altri device dell'ecosistema *Apple* (nello specifico Apple Watch[G]) per arricchire le informazioni mostrate a schermo dall'applicativo.

1.2 Il progetto

Il progetto di stage è consistito nello sviluppo di un'applicazione per *Apple TV*, parte di una piattaforma interna creata dall'azienda per la distribuzione e la visualizzazione di contenuti audio e video. Questa piattaforma è simile ai principali servizi di *streaming* presenti sul mercato, come *Netflix*, *Disney+*, *Amazon Prime Video*, ecc. . . , si differenzia però in base ai contenuti disponibili, in quanto, al contrario dei servizi di *streaming* che distribuiscono film o serie tv, in questo caso vengono distribuiti video relativi al fitness, più nello specifico video *workout*. Questi video sono registrazioni di allenamenti con dei *personal trainer*, che l'utente può seguire per allenarsi a casa.
Tramite l'app deve essere possibile accedere (scansionando un *QRCode*) alla piattaforma, navigarne i contenuti (tramite la navigazione della homepage, ricerca per genere o ricerca per titolo), riprodurre i contenuti (con anche la possibilità di riprendere da dove si era lasciato) e gestire la propria lista di preferiti. L'applicazione è stata sviluppata utilizzando il linguaggio *Swift* e il framework *SwiftUI*.
Successivamente è stato richiesto di esplorare e implementare un'eventuale integrazione con *HealthKit* e con altri device dell'ecosistema *Apple*. Per esempio, tramite l'utilizzo di *Apple Watch* è possibile raccogliere in tempo reale dati come il battito cardiaco e le calorie bruciate, che possono essere poi visualizzate durante la riproduzione del contenuto per tenere traccia dei propri progressi.

1.2.1 Obiettivi

Gli obiettivi principali previsti dal piano di lavoro stilato insieme all'azienda ospitante sono descritti di seguito, seguendo questa notazione:

- **O** per gli obiettivi obbligatori;
- **D** per gli obiettivi desiderabili;
- **F** per gli obiettivi facoltativi.

Obiettivi obbligatori

- **O1**: dimestichezza con Xcode[G] (IDE[G] di sviluppo per applicazioni iOS)
- **O2**: sviluppo di un applicativo in *Swift*
- **O3**: implementazione interazione con strutture server
- **O4**: interazione con *tvOS*

Obiettivi desiderabili

- **D1**: sviluppo di interfacce complesse
- **D2**: gestione AVPlayer[G] (SDK[G] *streaming* nativo)

- **D3**: proattività sul progetto
- **D4**: integrazione con *Apple Watch* e *HealthKit*

Obiettivi facoltativi

- **F1**: integrazione con strumenti e flusso di lavoro aziendale

1.3 Convenzioni tipografiche

Per quanto riguarda la stesura di questo documento, sono state adottate le seguenti convenzioni tipografiche:

- gli acronimi, le abbreviazioni e i termini ambigui o di uso non comune menzionati vengono definiti nel glossario, situato alla fine del documento;
- i termini in lingua straniera o facenti parte del gergo tecnico sono evidenziati con il carattere *corsivo*;
- tutti i termini appartenenti al glossario sono evidenziati e presentano una lettera G ad apice del termine presente (^{G}). Tale evidenziazione e dicitura sono presenti per la prima apparizione del termine all'interno del documento.

Capitolo 2

Architettura dell'applicativo

In questo capitolo verrà presentata l'architettura dell'applicativo sviluppato durante lo stage. In particolare, verranno presentati i componenti principali di tale architettura e le interazioni tra di essi.

2.1 Descrizione

L'applicativo per *tvOS* utilizza l'architettura interna all'azienda per lo sviluppo di app per *iOS*. Tale architettura è pensata come uno *starter kit* per ogni applicazione *iOS* dell'azienda, progettata per minimizzare il tempo di avvio di un nuovo progetto e per avere una struttura comune a tutte le app. Ciò permette a sviluppatori che lavorano su progetti diversi di avere comunque una conoscenza del funzionamento principale di ogni app, in modo da poter lavorare su più progetti senza dover gestire strutture fondamentalmente diverse. Inoltre, l'architettura è pensata per essere facilmente adattabile alla dimensione e complessità del progetto, ovvero è possibile utilizzare solo alcune parti di essa se le altre non sono necessarie per lo sviluppo.
Viene ripreso il pattern VIPER (*View*, *Interactor*, *Presenter*, *Entity*, *Router*) per la struttura, ma fortemente riadattato per essere in linea con le metodologie di sviluppo aziendali.

2.2 Strumenti e tecnologie utilizzate

L'architettura è stata pensata per utilizzare le ultime tecnologie per lo sviluppo nativo per piattaforme *Apple*. Questo permette di sviluppare applicativi che godono sempre delle ultime novità in termini di funzionalità derivanti dagli aggiornamenti software dei dispositivi *Apple*, e consente di ottenere anche sempre le ultime patch di sicurezza.

2.2.1 Linguaggi e framework

2.2.1.1 Swift

Il linguaggio di programmazione utilizzato è *Swift*, un linguaggio di alto livello sviluppato da *Apple* per la creazione di applicazioni per *iOS*, iPadOS[G], macOS[G], watchOS[G] e *tvOS*. *Swift* è orientato agli oggetti e fortemente tipizzato, ovvero richiede una dichiarazione di tipo per tutte le variabili e costanti. Nonostante la sicurezza dei tipi, è in grado di inferire automaticamente i tipi delle variabili e costanti quando possibile, riducendo la verbosità del codice. *Swift* presenta inoltre un sistema di gestione della memoria automatico, per gestirne l'allocazione e il rilascio, riducendo il rischio di memory leak[G] (fonte: [11]).

2.2.1.2 SwiftUI

Per lo sviluppo dell'interfaccia grafica è stato utilizzato *SwiftUI*, un *framework* per la creazione di interfacce utente per le piattaforme *Apple*. *SwiftUI* utilizza una sintassi dichiarativa, che permette di indicare cosa deve essere visualizzato, al contrario del predecessore UIKit[G], che utilizza una sintassi imperativa per descrivere come dovrebbero apparire gli elementi dell'interfaccia. *SwiftUI* è stato introdotto nel 2019, e per questo motivo non è ancora pienamente maturo. Tuttavia, è stato scelto per lo sviluppo dell'applicativo per *tvOS* per la sua semplicità e per la sua natura dichiarativa, che permette di scrivere meno codice rispetto a *UIKit* e di renderlo più leggibile. Nonostante ciò, alcuni componenti custom sono stati sviluppati in *UIKit*, che consente un maggiore controllo sulla visualizzazione dei componenti (fonte: [12]).

2.2.1.3 Altri framework

Sono state inoltre utilizzati altri *framework* nativi:

- **Combine**: *framework* per la gestione di flussi di dati asincroni ed eventi in *Swift*. È stato progettato per semplificare la gestione di dati reattivi e delle operazioni asincrone. Questo *framework* introduce in *Swift* i concetti di *Publisher* (che emettono sequenze di eventi, come valori di dati, errori e completamenti di azioni) e *Subscriber* (che ricevono e rispondono a tali eventi), oltre ad operatori di trasformazione e filtraggio dei flussi di eventi e gestione degli errori su tali flussi (fonte: [2]);
- **Core Data**: *framework* per la gestione di modelli di dati locali per applicazioni su piattaforme *Apple*. Permette di semplificare la gestione e la persistenza dei dati dell'applicativo, fornendo un'interfaccia ad alto livello per lavorare con il modello di dati senza dover scrivere manualmente il codice per l'accesso al database o la gestione della persistenza. Generalmente utilizza SQLite[G] come database sottostante (fonte: [3]).

2.2.2 Strumenti di sviluppo, versionamento e issue tracking

2.2.2.1 Xcode

Xcode è l'IDE ufficiale per lo sviluppo di applicativi per piattaforme *Apple*. È sviluppato da *Apple* ed è l'unico modo di compilare applicativi per *iOS* e le altre piattaforme e distribuirli su App Store[G]. *Xcode* è disponibile solo per *macOS*, rendendo lo sviluppo di tali applicativi possibile solo tramite l'utilizzo di un Mac[G]. Questo IDE presenta le comuni funzionalità degli ambienti di sviluppo integrati, come editor di testo, debugger[G], strumenti di analisi delle prestazioni, simulatori e supporta anche altri linguaggi oltre a quelli nativi *Apple*. Presenta anche alcune funzionalità generalmente non presenti sugli IDE più conosciuti, come una *preview* in tempo reale dell'applicazione in sviluppo, permettendo di visualizzare immediatamente le modifiche apportate al codice senza dover ricompilare l'applicativo (fonte: [15]).

2.2.2.2 Git e GitLab

Per il controllo di versione ho seguito le linee guida aziendali, utilizzando Git[G], e più nello specifico *GitLab* per la gestione della repository[G]. *GitLab* è un servizio di *hosting* per *repository Git*, che permette di gestire il versionamento del codice e le issue[G] relative al progetto. Dà inoltre la possibilità di gestire le *pipeline* di CI/CD[G], che permettono di automatizzare il processo di compilazione e distribuzione dell'applicativo. La *repository* è inserita all'interno del gruppo privato dell'azienda, accessibile solo dai membri del team di sviluppo (fonti: [7] [9]).

2.2.2.3 Postman

Il progetto prevede che l'applicativo per *Apple TV* comunichi con un server tramite delle richieste Application Program Interface[G]. Per semplificare il processo di utilizzo e testing di tali richieste ho utilizzato *Postman*, un applicativo per lo sviluppo di API. *Postman* offre varie funzionalità che rendono più semplice e veloce lo sviluppo di API, come la possibilità di creare e inviare richieste API, riceverne e visualizzarne le risposte, organizzare le richieste in collezioni, gestire variabili e ambienti. Inoltre, permette di creare e gestire la documentazione per le API, oltre che condividere gli ambienti e le collezioni con altri utenti (fonte: [13]).

2.2.2.4 Fork

Fork è un *client Git* per *macOS*, che permette di gestire le *repository Git* e di effettuare operazioni di versionamento. *Fork* consente di gestire le *repository* locali e remote, di effettuare operazioni di *merge* e di risolvere i conflitti, di effettuare il *push* e il *pull* delle modifiche, di gestire le *issue* e di effettuare il *rebase*. Inoltre, dà la possibilità

visualizzare le modifiche apportate al codice tramite un'interfaccia grafica, semplificando il processo di revisione del codice (fonte: [6]).

2.2.2.5 Linear

Linear è un issue tracker[G], e permette di creare e gestire le *issue* relative al progetto. È possibile creare *issue*, gestirne lo stato (*backlog*, da fare, in svolgimento, in verifica, completate), assegnarle a dei progetti e a dei membri del team, aggiungere commenti e tag, ecc. Inoltre, consente di creare delle *board* per visualizzare le *issue* in modo più intuitivo, e di creare delle *checklist* per tenere traccia delle operazioni da svolgere per ogni *issue* (fonte: [8]). L'utilizzo di *Linear* è stata una scelta personale, per gestire le attività da svolgere per il progetto. Internamente l'azienda invece utilizza *Jira*, un altro *issue tracker*.

2.2.2.6 Simulatori

Per effettuare il *testing* degli applicativi senza dover ogni volta installare il codice su dispositivi fisici, *Xcode* mette a disposizione dei simulatori dei principali dispositivi *Apple* e delle principali versioni dei loro sistemi operativi. In modo specifico, durante il progetto ho utilizzato i simulatori di *Apple TV* e *Apple Watch* per eseguire e testare gli applicativi in sviluppo.

2.2.3 Dispositivi di test

Per testare gli applicativi in casi reali di utilizzo, l'azienda mi ha fornito dei dispositivi fisici dove eseguire le app sviluppate. Ho potuto utilizzare un'*Apple TV 4K* di ultima generazione e un *Apple Watch Series 6*, oltre ad un *iPhone 11 Pro Max* necessario per l'installazione delle app su *Apple Watch*. In questo modo ho avuto la possibilità di utilizzare le app che ho sviluppato sui dispositivi dove sarebbe realmente eseguita, fornendomi la possibilità di mettermi nella posizione dell'utilizzatore reale e di risolvere eventuali problemi di usabilità.

2.3 Struttura

L'architettura è divisa in quattro layers che comunicano tra di loro:

- System layer
- View layer
- Control layer
- Data layer

2.3.1 System layer

2.3.1.1 AppCore

AppCore implementa il protocollo *App*, un tipo che rappresenta la struttura e il comportamento dell'app. *AppCore* rappresenta l'*entry point* dell'applicazione, ovvero il punto di partenza dell'esecuzione dell'app.
Il protocollo fornisce un'implementazione di default del metodo `main()` che il sistema chiama per avviare l'applicazione.
Il corpo di tale funzione contiene viste conformi con il protocollo *Scene*, che rappresentano le schermate dell'applicazione. Ogni scena contiene la vista radice di una gerarchia di viste e ha un ciclo di vita gestito dal sistema. *SwiftUI* fornisce dei tipi concreti di scene per gestire scenari comuni, come visualizzare documenti o impostazioni.
AppCore contiene:

- lo stato (*AppState*) dell'app condiviso con tutte le scene;
- il *SystemEventsHandler*;
- le dipendenze (*AppDependencies*).

2.3.1.2 AppState

L'architettura presenta un'*AppState* centralizzato come singola fonte di verità e mantiene lo stato dell'intera applicazione, come dati e informazioni sull'utente, token[G] di autenticazione e stato di sistema (l'app è attiva/in primo piano, l'app è in background[G], ecc...).
L'*AppState* è mantenuto all'interno di *AppCore* e referenziato in *AppDependencies*, in modo da essere accessibile ovunque nella gerarchia di scene dell'app. Dato quindi che l'*AppState* è l'unica fonte di verità dell'app, le scene non contengono business logic[G], ma visualizzano informazioni in funzione dello stato globale. Inoltre *AppState* è un *ObservableObject*, di modo che le viste possano osservarne i cambiamenti e aggiornarsi di conseguenza.
AppState non necessita di conoscere nulla rispetto agli altri layers dell'architettura, e perciò non contiene nessuna *business logic*.

2.3.1.3 AppDependencies

AppDependencies è uno struct[G] che contiene un riferimento all'*AppState* e agli *interactors* dell'app. Lo *struct* implementa il protocollo `EnvironmentKey`, che permette di accedere ai valori come una struttura chiave-valore. Nello stesso file viene dichiarata un'estensione di `EnvironmentValues`, che consente di accedere ai valori tramite la proprietà `environment` di `View`, in quanto i valori di ambiente sono propagati nella gerarchia di viste. Così facendo le dipendenze vengono inserite nelle scene dell'app,

tramite il pattern dependency injection[G], in modo da poter accedere alle dipendenze in ogni punto della gerarchia di viste usando il property wrapper[G] `@Environment`.

2.3.1.4 Handlers

Gli *Handlers* sono speciali tipi di servizi creati e gestiti da *AppCore*. Essi sono responsabili di gestire eventi di sistema, come l'avvio dell'app, la ricezione di push notifications[G], ecc. In base al numero e al tipo di eventi da gestire, un'app può avere un singolo handler, più handler diversi o nessun handler.
Gli handler hanno accesso ad *AppDependencies*, permettendo loro di aggiornare l'*AppState* e invocare *business logic* tramite gli *interactor*.

2.3.2 View layer

2.3.2.1 Scenes e Views

Le *Scene* rappresentano le diverse schermate dell'applicazione e non contengono *business logic*. Le *Views* sono componenti visuali che rappresentano le diverse parti di una scena e sono tipi indipendenti, per poter essere riutilizzate in altre scene.
I *side effects* sono invocati da azioni dell'utente (come il *tap* di un pulsante) o da eventi relativi al ciclo di vita delle viste (come `onAppear`, che aziona un evento quando la vista appare sullo schermo) e sono trasmessi agli *Interactor*. *AppState* e *Interactors* sono nativamente propagati nella gerarchia delle viste con `@Environment`, tramite il paradigma *dependency injection*, come spiegato precedentemente.
Generalmente una vista contiene il suo stato, un riferimento al *router* e le dipendenze (per accedere ad *AppState* e agli *Interactors*).

2.3.2.2 AppRouter

AppRouter gestisce la navigazione tra le scene dell'applicazione, e allo stesso momento disaccoppia la vista genitore dalla vista figlia. In questo modo, la vista genitore non deve conoscere la vista figlia, ma solo il *router*. *AppRouter* è mantenuto all'interno di *AppCore* e iniettato nella `RootView` dell'applicazione, in modo da poter essere accessibile in ogni punto della gerarchia di viste.
Ogni rotta è rappresentata nell'enumerazione `AppRoute`, che contiene tutte le possibili destinazioni della navigazione, ed è mappata su una specifica vista nel metodo `getView()`.
La natura dichiarativa di *SwiftUI* rende difficile separare la navigazione dal view layer, però questo *router* fornisce metodi che permettono di eseguire azioni di *push* (passare ad una scena), *pop* (tornare alla scena precedente), *present* (visualizzare una scena come un *modal*), *dismiss* (azione di chiusura di un *modal* o di ritorno alla scena precedente) in modo semplice ed intuitivo.

2.3.3 Control layer

2.3.3.1 Interactors

Gli *Interactor* incapsulano la *business logic* per una specifica scena o un gruppo di scene. Insieme all'*AppState* forma il control layer, che è completamente indipendente rispetto alla presentazione e alle risorse esterne.
Gli *interactor* sono completamente *stateless* (non mantengono uno stato internamente) e fanno riferimento all'oggetto *AppState*, passato come parametro al costruttore tramite *dependency injection*. Deve essere presente un protocollo per ogni *interactor*, in modo da permettere alle viste di accedere ai metodi dell'*interactor* senza conoscere la sua implementazione e di utilizzare un *mocked interactor* (un *interactor* le cui funzioni ritornano valori di test) in fase di test.
Gli *interactor* ricevono richieste da parte delle scene per eseguire azioni, come ottenere dati da una fonte esterna o eseguire calcoli, però essi non ritornano mai dei dati direttamente. Inviano tali risultati all'*AppState* se tali dati devono essere accessibili a tutte le viste e se non sono molti (per esempio l'email di login), o li ritornano indirettamente alla vista che ha fatto la richiesta tramite un Binding[G], utile in caso ci siano grandi quantità di dati da ritornare.

2.3.4 Data layer

2.3.4.1 Services

I *Services* forniscono un'API asincrona (basata su *async/await*) per effettuare operazioni CRUD[G] sul *backend* o su un database locale. Questi servizi non contengono *business logic* né modificano l'*AppState*.
I *services* sono accessibili e utilizzabili solo dagli *interactors* e sono nascosti dietro un protocollo per permettere ad essi di usare un *mocked service* durante i test. I *services* sono inizializzati in *AppCore*, però siccome il loro riferimento è mantenuto dall'*interactor* che lo utilizza, se un *interactor* non utilizza un service, esso non viene inizializzato e non spreca risorse di sistema. I *Managers* sono servizi da cui altri servizi dipendono. Seguono le stesse regole dei normali *services* ma sono inizializzati prima di essi, per permettere la *dependency injection*. Un esempio è `SessionManager`, che gestisce la sessione *HTTP* ed è usato dal servizio che gestisce le API di connessione al *backend*, per autenticarne le richieste.

Capitolo 3

Design e sviluppo dell'applicativo

In questo capitolo verrà presentato il processo di design e sviluppo dell'applicativo, in particolare verrà illustrato come sono stati progettati e implementati i principali componenti e strutture dell'applicativo, insieme a porzioni di codice significative.

La fase di codifica dell'applicativo si è divisa in tre parti:

- **Sviluppo dell'interfaccia utente**: in questa fase sono stati implementati tutti i componenti grafici dell'applicativo, in particolare sono stati implementati i *layout* delle varie pagine e i componenti grafici che li compongono, come ad esempio i pulsanti, le liste, le immagini, ecc.
- **Sviluppo della business logic**: in questa fase sono stati implementati tutti i componenti che gestiscono la logica dell'applicativo, come ad esempio la gestione del login, la gestione dei contenuti, la gestione dei preferiti, ecc. Queste operazioni sono svolte dagli *interactors*.
- **Sviluppo della comunicazione con il server**: in questa fase è stato implementato il codice che permette la comunicazione con il *backend* della piattaforma, in particolare è stata implementata una libreria interna che permette di effettuare le richieste al server e di gestire le risposte.

3.1 Sviluppo dell'interfaccia utente

Questa sezione descrive come sono stati implementati i componenti grafici dell'applicativo, indicando le scelte di design fatte e le motivazioni che hanno portato a tali scelte.

3.1.1 Linee guida

Non essendo presenti dei mockup[G] aziendali per l'applicativo per *tvOS*, ho seguito le linee guida per lo sviluppo di app *tvOS* (fonte: [4]) di *Apple*.
Ho utilizzato principalmente componenti nativi, per ridurre al minimo eventuali errori o *bug* dovuti all'interfaccia, inserendo in caso di necessità alcuni componenti custom, per mantenere l'esperienza di navigazione simile all'app *iOS* già realizzata dall'azienda e ovviare a qualche mancanza nei componenti nativi *Apple*.

3.1.2 Requisiti

Insieme al tutor aziendale, sono stati individuati i seguenti requisiti per l'interfaccia utente dell'applicativo:

- L'app deve permettere di effettuare il login tramite la scansione di un QRCode[G] con un dispositivo mobile;
- L'app deve permettere di navigare i contenuti della piattaforma, suddivisi nelle categorie previste dal *backend*;
- L'app deve permettere di effettuare ricerche sui contenuti in base al loro titolo;
- L'app deve permettere di visualizzare i dettagli di un contenuto, come il titolo, la descrizione e durata;
- L'app deve permettere di riprodurre un contenuto, e di riprenderne la visione dal punto in cui era stata interrotta;
- L'app deve permettere di visualizzare e gestire i contenuti preferiti dell'utente, dando la possibilità di aggiungere o rimuovere un contenuto dai preferiti;
- L'app deve permettere di visualizzare il profilo dell'utente, con i dettagli del suo piano sottoscritto e la possibilità di effettuare il logout.

3.1.3 Struttura

Come ispirazione ho utilizzato le app *mobile* e per Smart TV[G] delle principali piattaforme di streaming[G] (esempio in figura 3.1, fonte: [10]), che presentano interfacce tutte simili tra di loro, e l'app *iOS* già sviluppata dall'azienda per la piattaforma interna. Le app di *streaming* presentano tutte una struttura simile, con una homepage che visualizza i contenuti più popolari e le categorie principali, una pagina per indicare i contenuti salvati o preferiti, una pagina di ricerca per categoria o nome del contenuto e una pagina utente per gestire l'account e le impostazioni. Inoltre ogni contenuto ha una pagina di dettaglio, con informazioni aggiuntive e la possibilità di avviare la riproduzione, o riprenderla in caso fosse già stata avviata in precedenza.

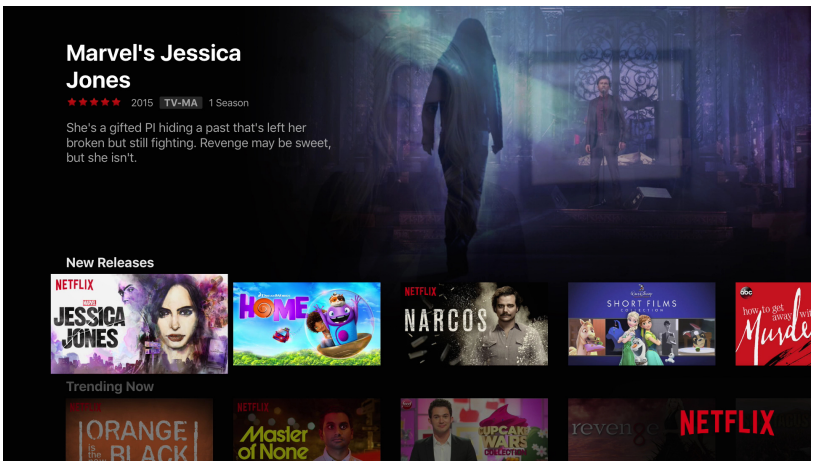


Figura 3.1: App Netflix per Apple TV

In base a quanto appena descritto e ai requisiti individuati insieme al tutor aziendale, ho deciso di strutturare l'applicativo nelle seguenti nove pagine:

- Home
- Musica
- Generi
- Pagina di dettaglio del contenuto
- Pagina del video player
- Pagina dei preferiti
- Pagina di ricerca
- Pagina utente
- Pagina di login

3.1.3.1 Navigazione principale

Per la navigazione tra le diverse pagine dell'app, ho utilizzato un'interfaccia a schede (figura 3.2), usando componenti nativi di *tvOS* per gestire tale navigazione. Ho scelto la divisione in schede perché è la modalità principale di navigazione tra pagine nelle app *tvOS*, in quanto è particolarmente intuitiva per l'utente, che deve navigare l'interfaccia utilizzando un telecomando e non un *touch screen*.

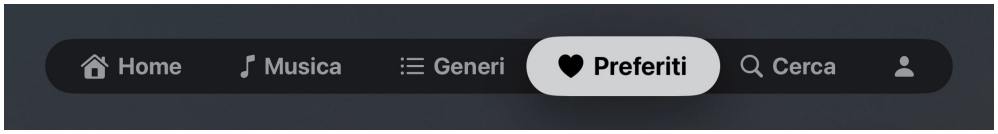


Figura 3.2: Apple TV — Interfaccia di navigazione

3.1.3.2 Pagina di login

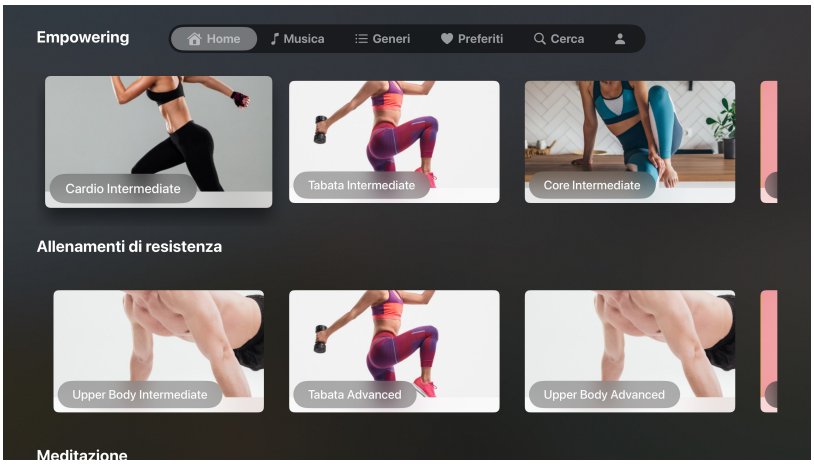
La pagina di login (figura 3.3) permette ad un utente di effettuare l'accesso alla piattaforma.
In base ai requisiti, era richiesto che il login non avvenisse tramite *username* e *password*, ma tramite la scansione di un *QRCode* visualizzato sullo schermo della tv tramite un dispositivo mobile. La pagina visualizza il codice QR insieme alle istruzioni su come effettuare il login e un codice di 6 cifre associato al dispositivo *Apple TV*. Il *QRCode* porta l'utente alla pagina web di accesso alla piattaforma, dove può effettuare il login o la registrazione in caso non abbia ancora creato un *account*. Durante il login, la pagina web richiederà di inserire il codice di 6 cifre visualizzato sulla tv, in modo da poter associare l'operazione di login con lo specifico dispositivo *Apple TV*. Una volta effettuato il login tramite la pagina web, l'app per *Apple TV* viene automaticamente aggiornata e l'utente è reindirizzato alla homepage.

Figura 3.3: Apple TV — Pagina di login

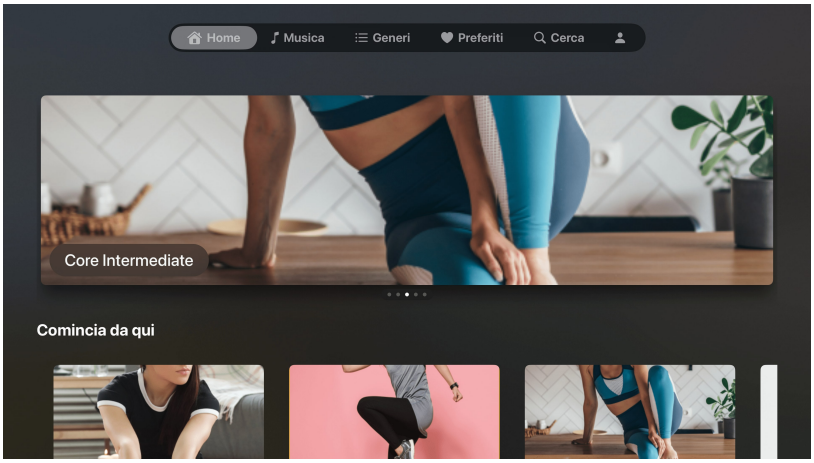
(Nella figura 3.3 il codice QR e il link di accesso sono stati oscurati per motivi di riservatezza dell'azienda).

3.1.3.3 Homepage

La homepage (figura 3.4) è il primo punto di accesso all'applicativo da parte dell'utente. In questa pagina vengono mostrati i contenuti più popolari o eventualmente consigliati dalla piattaforma, e le categorie principali di contenuti. La navigazione è gestita tramite dei caroselli, ovvero delle liste che scorrono in orizzontalmente, ottimali per gli schermi dei televisori. Questi sono poi visualizzati uno dopo l'altro, in una sequenza che si sviluppa dall'alto verso il basso (figura 3.4a). Ognuno dei caroselli identifica una categoria di contenuti o un insieme di contenuti consigliati. È presente anche un carosello di tipo *Hero*[G], che visualizza un solo contenuto alla volta, in modo da dare maggiore risalto ad esso (figura 3.4b). È utilizzato per contenuti particolarmente popolari o consigliati dai curatori della piattaforma.



(a) Caroselli orizzontali



(b) Carosello Hero

Figura 3.4: Apple TV - Homepage

3.1.3.4 Musica

Alla pagina Musica si accede selezionando la scheda Musica nella barra di navigazione principale. Essa visualizza tutti i contenuti audio presenti nella piattaforma, suddivisi per categoria. La navigazione avviene come nella homepage, tramite dei caroselli orizzontali che visualizzano tutti i contenuti di una categoria. Al momento dello sviluppo dell'app, non erano presenti contenuti audio nella piattaforma, quindi la pagina visualizza solo un messaggio che indica che nessun contenuto è presente.

3.1.3.5 Generi

La pagina Generi (figura 3.5) è accessibile selezionando la scheda Generi nella barra di navigazione principale. Essa visualizza tutti i generi e le categorie di contenuti presenti

nella piattaforma. Come per la pagina home, i contenuti sono visualizzati tramite caroselli orizzontali, ognuno dei quali identifica un genere o una categoria.
In questo caso, siccome lato *backend* non sono state definite categorie specifiche da visualizzare nella home, i contenuti di questa pagina e della home sono gli stessi. Per questo motivo anche in questa pagina è presente il carosello *Hero*.

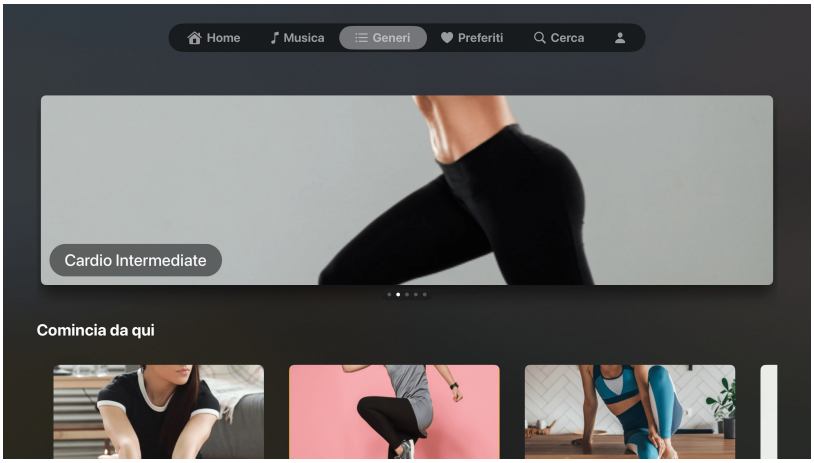


Figura 3.5: Apple TV - Pagina Generi

3.1.3.6 Pagina di dettaglio del contenuto

La pagina di dettaglio del contenuto (figura 3.6) è accessibile cliccando su un contenuto presente nella homepage o in una delle altre pagine. In questa pagina viene descritto in modo più dettagliato il contenuto, visualizzandone il titolo, una thumbnail[G], la descrizione e la durata. Da questa pagina è inoltre possibile avviare la riproduzione del contenuto e anche aggiungerlo ai preferiti.
La pagina presenta inoltre anche un tasto per avviare una connessione con *Apple Watch*, in modo da poter poi visualizzare i dati biometrici in tempo reale durante un video *workout*. Il tasto è identificato dall'icona di *Apple Watch* e viene visualizzato solo se l'*Apple TV* utilizzata è compatibile con il protocollo di comunicazione descritto nel capitolo 4.

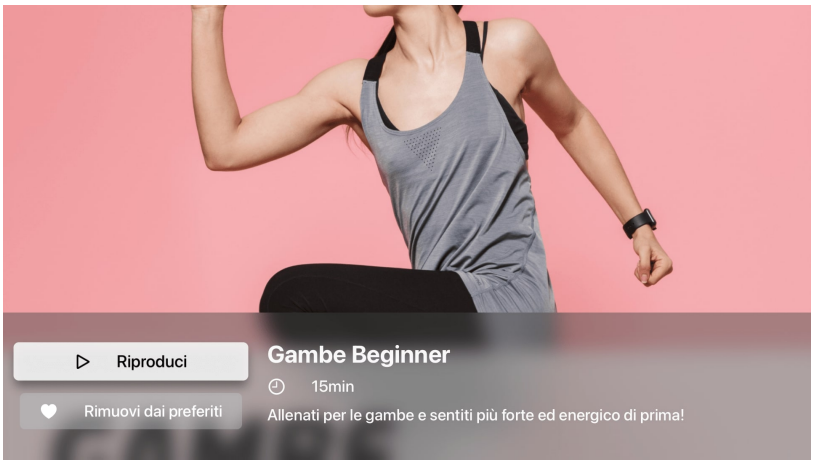


Figura 3.6: Apple TV — Pagina di dettaglio del contenuto

3.1.3.7 *Video player*

Il *video player* (figura 3.7) è il componente che visualizza il contenuto selezionato dall'utente. Come decisione progettuale è stato scelto di utilizzare il *video player* nativo di *tvOS*, in quanto permette ad un utente di *Apple TV* di ottenere la stessa esperienza di visualizzazione che avrebbe in ogni altra app *tvOS*, in modo da non dover imparare ad utilizzare una nuova interfaccia. Da questo componente è possibile avviare e mettere in pausa la riproduzione, eseguire azioni di *fast forward* e *rewind*, e visualizzare la durata del contenuto e il tempo trascorso.
Ho inoltre implementato la funzionalità che permette di riprendere la visione del contenuto dal punto dove lo si era lasciato durante la visione precedente, in modo da rendere l'esperienza di visione più fluida e simile a quella che si ha su altre piattaforme. Il funzionamento è il seguente:

- L'utente avvia la riproduzione di un contenuto, ma non lo visualizza fino alla fine, e chiude il *video player*;
- L'utente torna a visualizzare lo stesso contenuto in un secondo momento, anche dopo aver riavviato l'app, e apre il *video player*;
- Il video viene automaticamente riprodotto dal punto in cui era stato interrotto nella visione precedente.

Questa funzionalità era già presente lato *backend* con una chiamata, dove è possibile salvare e recuperare l'ultimo tempo di visione del contenuto. Il salvataggio del tempo di visione sul *backend* avviene al momento della chiusura del *video player*, mentre il recupero avviene al momento dell'apertura dello stesso. Una volta recuperato il tempo di visione, viene effettuata un'operazione di seek[G] sul *player* in modo da portare il video al punto corretto.

Durante lo sviluppo è stato usato un video di test per verificare il funzionamento del *video player*.

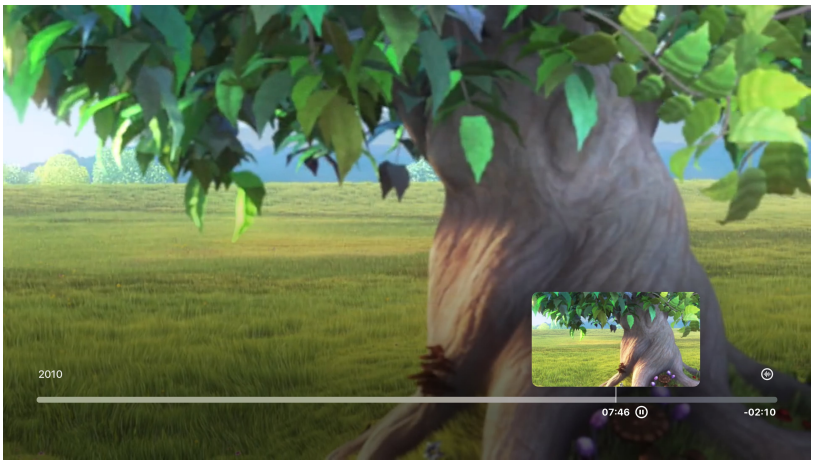


Figura 3.7: Apple TV — Video Player

3.1.3.8 Pagina dei preferiti

La pagina dei preferiti (figura 3.8) raccoglie tutti i contenuti che l'utente ha indicato come preferiti, sia video che audio. Il design di tale pagina riprende quello della homepage: sono presenti due caroselli, uno per i contenuti video e uno per i contenuti audio, con la possibilità di scorrere orizzontalmente i contenuti e di selezionarli per visualizzarne i dettagli. Cliccare su uno qualsiasi dei contenuti preferiti porta alla pagina di dettaglio del contenuto.

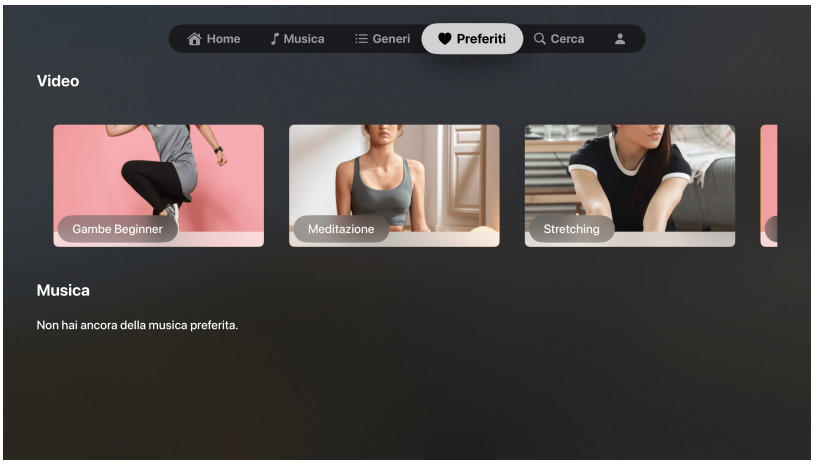


Figura 3.8: Apple TV — Pagina dei preferiti

3.1.3.9 Pagina di ricerca

La pagina di ricerca (figura 3.9) permette di cercare i contenuti in base al loro titolo. L'interfaccia usa un componente nativo di *tvOS*, che visualizza una barra di ricerca *fullscreen* insieme ad una tastiera virtuale, in quanto il telecomando di *Apple TV* non presenta tasti alfanumerici, ma solo frecce direzionali e un *touchpad*. Tramite la tastiera virtuale l'utente è in grado di inserire le parole chiavi da cercare, e in tempo reale vengono visualizzati i contenuti pertinenti tramite un carosello orizzontale come quelli descritti in precedenza. Essendo il componente di ricerca nativo, è automaticamente compatibile anche con la ricerca tramite input vocale, tramite l'apposito tasto presente sul telecomando. In questo modo anche per gli utenti con disabilità motorie è possibile effettuare una ricerca in modo semplice e veloce.

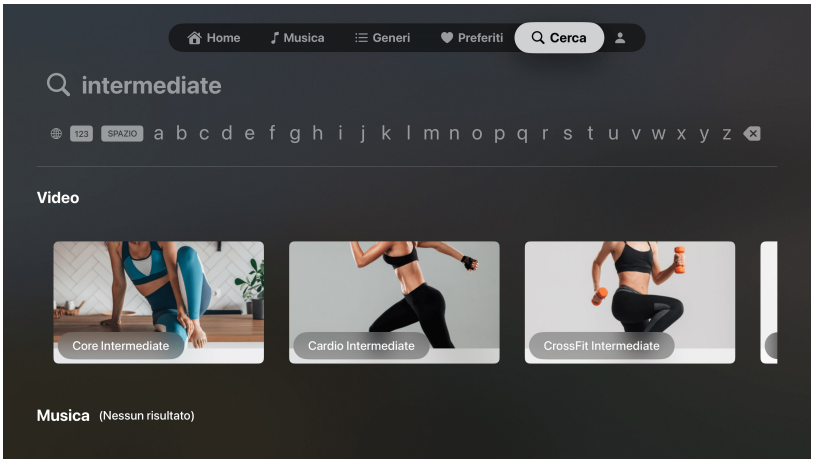


Figura 3.9: Apple TV — Pagina di ricerca

3.1.3.10 Pagina utente

La pagina utente (figura 3.10) visualizza l'attuale utente che sta utilizzando l'applicazione e i dettagli del suo profilo. In questa pagina viene mostrata la foto profilo dell'utente, se esso ne ha impostata una, il suo nome utente se l'ha impostato (o se non l'ha fatto l'email) e i dettagli del piano sottoscritto. Questi dettagli includono il nome del piano, il prezzo e la data di scadenza. Inoltre è presente anche un pulsante per effettuare il logout, che richiede una conferma e successivamente porta l'utente alla pagina di login.

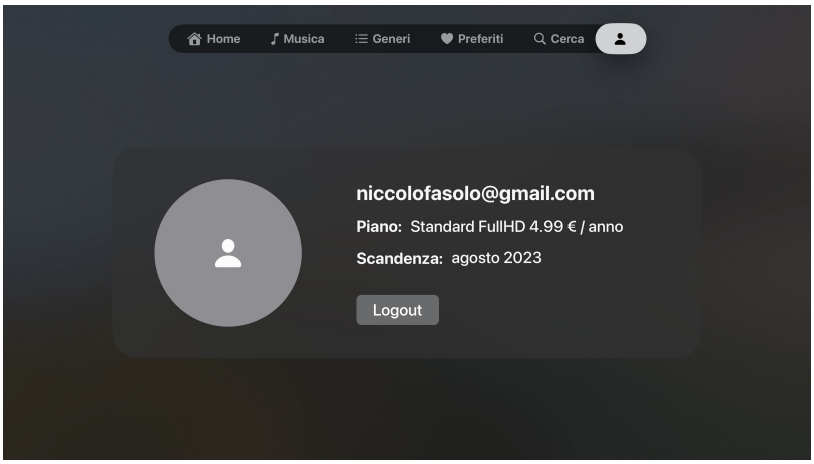


Figura 3.10: Apple TV — Pagina del profilo

3.2 Sviluppo della comunicazione con il backend

Per accedere ai dati della piattaforma di *streaming*, è stato necessario sviluppare un sistema di comunicazione con il *backend*. Quest'ultimo espone già degli *endpoint* a cui è possibile effettuare delle richieste *HTTP* per ottenere i dati necessari. È stato quindi necessario sviluppare un sistema che permettesse di effettuare queste richieste e di gestire le risposte, in modo da poterle utilizzare all'interno dell'applicazione.

3.2.1 *AppWebService*

Per sviluppare il sistema di comunicazione, ho utilizzato il *template* presente all'interno dell'architettura aziendale, ovvero il protocollo *AppWebService* (listing 3.1). Il protocollo espone i metodi necessari per effettuare le chiamate al *backend*.
Per utilizzare questo protocollo, ho creato un'implementazione di esso nella classe *RealAppWebService*, fornendo anche le implementazioni di tutti i metodi che effettuano tali chiamate.

```
1   public protocol AppWebService: WebService {
2       func getConfig() async throws -> Config
3       func getPage(with id: String) async throws -> Page
4       func getComponent(with id: String,
5                         offset: Int) async throws -> Component
6       func getVideoItem(with id: String) async throws -> Item
7       func getAudioItem(with id: String) async throws -> Item
8       func isFavorite(with contentId: String) async throws ->
    IsFavoriteResponse
9       func getFavorites() async throws -> [FavouriteItem]
10      func removeFavorite(id: String) async throws
11      func addFavorite(item: NewFavouriteItem) async throws ->
    FavouriteItem
12      func getWatchTime(ofVideo id: String) async throws -> WatchTime
13      func updateWatchTime(ofVideo id: String,
14                           item: NewWatchTime) async throws
```

```
15          func search(with key: String) async throws -> [Component]
16          func getUser() async throws -> User
17          func getSubscription() async throws -> Subscription
18          func getTVCode() async throws -> String
19          func getJWTToken(for tvCode: String,
20                           checksum: String,
21                           salt: String) async throws -> String
22      }
```

Listing 3.1: Protocollo *AppWebService* per la comunicazione con il backend

Ogni metodo effettua la chiamata al *backend*, e se questa va a buon fine, vengono ricevuti i risultati in formato JSON[G]. Se invece la chiamata non va a buon fine, viene lanciata un'eccezione che può essere gestita dal chiamante.
I dati ricevuti sono poi decodificati usando il *decoder JSON* nativo di *SwiftUI* e vengono restituiti al chiamante sotto forma di oggetto.

3.2.2 Autenticazione

Per garantire l'accesso al *backend* solo da parte di utenti autorizzati, le chiamate effettuate devono essere autenticate. Per fare ciò, è stato necessario implementare un sistema di autenticazione che permettesse di ottenere un *token* di accesso da utilizzare nelle chiamate successive.
Al momento del login nell'app, al dispositivo *Apple TV* viene assegnato un codice numerico di 6 cifre, che agisce da identificatore per il dispositivo. Questo codice è utilizzato nella fase di login su dispositivo mobile per associare il dispositivo al profilo dell'utente.
Quando viene fatta l'associazione, *Apple TV* riceve un primo token JWT[G] , che permette all'app di effettuare alcune chiamate al *backend*. Questo token è suddiviso in tre altri *token*:

- **Access token**: è il *token* di autorizzazione, che permette di autenticare tutte le chiamate al *backend*;
- **Refresh token**: è il *token* che permette di rinnovare l'*access token*;
- **ID token**: è il *token* che contiene i dati dell'utente.

Ognuno di questi tre *token* ha una durata predefinita, e viene salvato in locale, all'interno del Keychain[G] di Apple, per poter essere utilizzato per le chiamate successive.
L'*Access token* è quindi passato nell'*header* di ogni chiamata, in modo da autenticarla e permetterne l'esecuzione.
Una volta scaduto l'*Access token*, le chiamate al *backend* vengono rifiutare con un errore di autenticazione (401 *unauthorized*), ed è necessario effettuare il rinnovo dell'*Access token*.

3.2.2.1 Rinnovo dell'*Access token*

Per effettuare il rinnovo dell'*Access token*, viene effettuata una chiamata al *backend*, autenticata con il *token* scaduto, e con all'interno del *body* della richiesta il *Refresh token* salvato in precedenza nel *Keychain*.
Se la chiamata va a buon fine, viene restituito un nuovo *Access token*, salvato all'interno del *Keychain* al posto del precedente. Viene inoltre eseguita nuovamente la chiamata che era stata precedentemente rifiutata a causa del *token* scaduto.

3.2.3 Libreria interna di comunicazione

Il codice di comunicazione con il *backend* appena descritto è stato fino a questo punto parte della codebase[G] del progetto. Durante lo sviluppo però, l'azienda ha deciso che per le app per piattaforme *Apple* sarebbe stato più utile esportare tale codice in una libreria esterna, in modo da poter essere riutilizzato in tutte le app dell'ecosistema *Apple* che accedono alla piattaforma di *streaming* interna. È stato quindi necessario, insieme al tutor aziendale, creare tale libreria di comunicazione.
Siccome l'app per *iOS* e *tvOS* utilizzavano paradigmi diversi per effettuare le chiamate al *backend* (*Publishers* su *iOS* e *async/await* su *tvOS*), è stato necessario sceglierne uno comune per entrambe, e creare la libreria sulla base di tale paradigma. Il paradigma scelto è stato quello utilizzato nell'app per *tvOS*, ovvero *async/await*, in quanto più nuovo e più adatto ad effettuare tale comunicazione.
Da qui ho poi effettuato il *porting* del codice di comunicazione all'interno della libreria, e ho modificato il codice dell'app per utilizzarla, importandola tramite Swift Package Manager[G]. In questo modo è possibile eseguire delle modifiche alla libreria di comunicazione ed estenderne le funzionalità, senza dover modificare il codice dell'app.

3.3 Sviluppo della *business logic*

La *business logic* dell'app è la parte che si occupa di eseguire l'effettiva computazione di dati. In questo caso, la logica è suddivisa per scopo in diversi *interactors* (come descritto nel capitolo 2). In base alle funzioni svolte, possiamo suddividere la logica in tre categorie principali:

- **Gestione del login**: si occupa di gestire l'accesso dell'utente all'app e il recupero dei suoi dati;
- **Gestione dei contenuti**: si occupa di gestire il recupero dei contenuti della piattaforma di *streaming*, gestisce inoltre la suddivisione dei contenuti in categorie e generi e la ricerca dei contenuti;
- **Gestione dei preferiti**: si occupa di gestire la lista di contenuti preferiti dell'utente e le azioni di aggiunta e rimozione da essa;

- **Gestione dei contenuti in riproduzione**: si occupa di gestire i contenuti in riproduzione nel video player, e della funzionalità di *resume* della riproduzione.

Nota: alcuni dei metodi mostrati successivamente fanno utilizzo del tipo generico `Loadable<T>`, che è un enumerazione[G] che rappresenta un dato che si può trovare in uno di quattro stati: *notRequested*, *isLoading*, *loaded* e *failed*. Questo tipo è stato utilizzato per gestire lo stato di caricamento dei dati, in modo da poter mostrare all'utente un indicatore di caricamento, e in caso di errore mostrare un messaggio informativo.

3.3.1 Gestione del login

La gestione del login è svolta da *UserInteractor* (listing 3.2), che gestisce tutte le azioni che riguardano l'accesso dell'utente e la gestione dei suoi dati personali.

```
1    protocol UserInteractor {
2        func getTVCode(result: Binding<Loadable<String>>)
3        func startPollingForToken(numberOfAttempts: Int?) -> Task<Void,
     Error>
4        func trySignIn() -> Bool
5        func getUserData()
6        func getSubscription(result: Binding<Loadable<Subscription>>)
7        func logout()
8    }
```

Listing 3.2: Protocollo *UserInteractor* per la gestione dell'accesso e dei dati personali dell'utente

3.3.1.1 Login

L'operazione di login si articola in varie fasi:

- Prima di tutto, viene richiesto al *backend* in modo non autenticato, tramite il metodo `getTVCode(result:)`, un codice univoco di 6 cifre che viene associato alla specifica *Apple TV*;
- Viene quindi visualizzata la pagina di login, che mostra un QRCode che, una volta scansionato tramite un dispositivo mobile, porta alla pagina web di login del servizio di streaming. Quando la pagina viene visualizzata, l'app inizia un'operazione di polling[G] al *backend*, ogni secondo, per verificare se l'utente ha effettuato il login. L'operazione è avviata tramite il metodo `startPollingForToken(numberOfAttempts:)`, che richiede in modo opzionale il numero di tentativi di *polling* da effettuare. Se il numero di tentativi non viene specificato, l'operazione viene eseguita in modo indefinito. Questa chiamata invia al *backend* una APIKEY[G] crittografata tramite AES[G] che contiene il codice della tv generato precedentemente. La generazione di tale *APIKEY* è spiegata successivamente. Durante il *polling*, il controllo del completamento del login è fatto richiedendo al *backend* il *token JWT* che serve all'app per effettuare le successive richieste autenticate. Se la

richiesta ritorna un errore 404 (*not found*) significa che l'utente non ha ancora completato l'operazione di login;

- Nella pagina web viene richiesto all'utente di effettuare il login tramite le sue credenziali (o viene richiesto di effettuare la registrazione, in caso l'utente non abbia un *account*);
- Successivamente, viene richiesto il codice univoco di 6 cifre che è stato generato in precedenza, e che è mostrato all'interno dell'app, in modo da abbinare l'*account* dell'utente all'*Apple TV*;
- Una volta completato l'abbinamento, l'app su *Apple TV* che finora è rimasta in *polling*, smette di ricevere risposte con errore 404 e riceve invece il *token JWT* con cui autenticare le chiamate successive. A questo punto, l'operazione di *polling* termina e l'utente viene reindirizzato alla schermata principale dell'app.
- Quando il login è completato, il codice univoco della tv viene salvato nel database UserDefaults[G] dell'app, in modo da poterlo recuperare alla successiva apertura. In questo modo, quando l'app viene riaperta, le operazioni appena descritte vengono ripetute, questa volta tramite il metodo `trySignIn()`, però non appena viene avviato il *polling*, viene subito ricevuto il *token JWT*, in quanto il *backend* lo mantiene salvato e non richiede che l'utente effettui nuovamente il login tramite dispositivo mobile. L'utente viene quindi reindirizzato alla schermata principale dell'app.

3.3.1.2 Generazione dell'*APIKEY*

L'*APIKEY* viene generata e passata al *backend* durante il *polling* per evitare di condividere il codice della tv in chiaro.
l'*APIKEY* è creata concatenando la data e ora attuale con il codice della tv e un *salt*, ovvero una stringa univoca per l'app. Questa stringa viene poi crittografata tramite l'algoritmo *AES*, che utilizza una chiave segreta condivisa tra *backend* e *frontend*.
Durante lo sviluppo di questa funzionalità, ho incontrato un problema: l'implementazione dell'algoritmo di crittografia *AES* del linguaggio *Swift*, accessibile tramite la libreria nativa *CommonCrypto*, non è compatibile con l'algoritmo *AES* utilizzato dal *backend*, implementato tramite la libreria *CryptoJS*. Questo comporta che la stringa crittografata dall'app non venga decifrata correttamente dal *backend*.
Dopo alcuni tentativi di risoluzione del problema, inizialmente erroneamente associato al tipo di codifica della stringa ritornata dall'algoritmo, l'unica soluzione funzionante trovata è stata quella di implementare all'interno di *Swift* l'algoritmo *AES* tramite *CryptoJS*. Questo è stato fatto tramite l'utilizzo della libreria nativa di *Swift JavaScriptCore*, che permette di creare un ambiente di esecuzione di codice *JavaScript* all'interno di *Swift*. Ho quindi importato l'intera libreria *CryptoJS* all'interno del

progetto e ho creato una funzione *JavaScript* che esegue l'algoritmo di crittografia *AES* e ritorna la stringa crittografata. Ho quindi successivamente eseguito un test di funzionamento, utilizzando la stringa crittografata durante l'operazione di *polling*, che ha avuto esito positivo.

3.3.1.3 Gestione dei dati utente e del piano sottoscritto

Questo *interactor* esegue anche le operazioni di richiesta al *backend* dei dati dell'utente e del piano sottoscritto. Questi dati vengono visualizzati nella pagina del profilo dell'utente, che viene mostrata quando l'utente sceglie la scheda del profilo nella schermata principale dell'app.
I dati sono richiesti tramite i metodi `getUserData()` e `getSubscription(result:)`. Questi metodi non ritornano direttamente i dati, ma li inseriscono nello stato globale dell'app, in modo da essere accessibili ovunque in caso possano ritornare utili in un momento successivo.
I dati dell'utente che vengono richiesti sono:

- Email;
- *UUID* (un codice identificativo univoco);
- Nome e cognome (opzionali, ammettono il valore `null` in caso non siano stati impostati);
- Immagine del profilo (opzionale, ammette il valore `null` in caso non sia stata impostata).

I dati del piano sottoscritto che vengono richiesti sono:

- Nome del piano sottoscritto;
- Prezzo;
- Data di scadenza.

3.3.2 Gestione dei contenuti

La gestione dei contenuti è fatta da *PagesInteractor* (listing 3.3), che gestisce il fetch[G] dei contenuti e la loro suddivisione in generi, categorie e pagine (chiamate *pages*), oltre che la ricerca.

```
1     protocol PagesInteractor {
2         func load(pages: Binding<Loadable<[Page]>>)
3         func load(components: Binding<Loadable<[Component]>>,
4                   forPageWith id: String)
5         func load(items: Binding<Loadable<[Item]>>,
6                   forComponentWith _: String,
```

```
7                   offset: Int)
8         func loadVideo(item: Binding<Loadable<Item>>)
9         func search(components: Binding<Loadable<[Component]>>,
10                    for string: String)
11    }
```

Listing 3.3: Protocollo *PagesInteractor* per la gestione dei contenuti e la ricerca

3.3.2.1 *Fetch* delle pagine

Il *fetch* delle pagine viene eseguito tramite il metodo `load(pages:)`, che ritorna un array che identifica le pagine principali dell'app: Home, Musica e Generi. Queste pagine sono identificate da un *id* univoco, che viene salvato nello stato globale dell'app per le chiamate successive. Inoltre viene ritornata la lista di queste pagine in quanto è possibile rimuovere pagine da questa lista lato *backend* e automaticamente queste non verranno visualizzate nell'app.

3.3.2.2 *Fetch* dei componenti

Tramite gli identificatori delle pagine salvati nello stato globale dell'app, è possibile richiedere i componenti che le compongono, utilizzando il metodo `load(components: forPageWith:)`. Questo metodo ritorna dei *Components*, che identificano i componenti di una pagina, in questo caso i caroselli orizzontali presenti nelle pagine Home, Musica e Generi.

3.3.2.3 *Fetch* degli *Items*

Ogni componente è poi composto da un insieme di *Items*, che identificano un singolo contenuto, sia questo video o audio. Questi vengono richiesti tramite il metodo `load(items:forComponentWith:offset:)`. Questa chiamata ritorna un *array* di *Items* che vengono poi visualizzati all'interno del componente, e richiede di fornire un *offset*, in quanto, dato che un componente può essere composto da molti *Items*, la chiamata è paginata, ovvero ad ogni chiamata viene ritornato un numero limitato di *Items*, e l'*offset* indica da quale *Item* iniziare a ritornare i risultati.

3.3.2.4 *Fetch* di un singolo *Item*

Questo *interactor* inoltre gestisce il caricamento dei dati di un video, ovvero in questo caso dei dettagli di un singolo *Item*, tramite il metodo `loadVideo(item:)`. Questo metodo ritorna un *Item* che viene poi visualizzato nella pagina di dettaglio del contenuto.

3.3.2.5 Ricerca

Infine questo *interactor* gestisce anche la ricerca dei contenuti, tramite il metodo `search (components:for:)`. Questo metodo richiede una stringa, passata *backend* e utilizzata

per ricercare dei *match* sui titoli dei contenuti. Viene quindi ritornato un *array* di *Components*, che conterrà sempre due componenti: uno per i video e uno per l'audio. Questi componenti vengono poi visualizzati nella pagina di ricerca, che viene mostrata quando l'utente sceglie la scheda di ricerca nella schermata principale dell'app.

3.3.3 Gestione dei preferiti

La gestione dei preferiti è fatta da *FavoriteInteractor* (listing 3.4), che gestisce il *fetch* dei contenuti preferiti da visualizzare in lista e la loro aggiunta o rimozione dai preferiti.

```
1    protocol FavoriteInteractor {
2        func loadFavorite(vitems: Binding<Loadable<[Item]>>,
3                          aitems: Binding<Loadable<[Item]>>)
4        func addFavorite(item: Item, type: FavoriteType)
5        func removeFavorite(id: String)
6        func isFavoriteItem(item: Item, result: Binding<Bool>)
7    }
```

Listing 3.4: Protocollo *FavoriteInteractor* per la gestione dei preferiti

3.3.3.1 *Fetch* della lista dei preferiti

Il *fetch* della lista dei preferiti viene fatto tramite il metodo `loadFavorite(vitems: aitems :)`. Il metodo ritorna due *array*, uno per i contenuti video preferiti e uno per i contenuti audio preferiti. Questi *array* vengono poi visualizzati nella pagina dei preferiti, che viene mostrata quando l'utente sceglie la scheda dei preferiti nella schermata principale dell'app.

3.3.3.2 Aggiunta e rimozione di un contenuto dai preferiti

Per aggiungere un contenuto ai preferiti, viene utilizzato il metodo `addFavorite(item :type:)`. Questo metodo richiede un *Item* e un *FavoriteType*, che indica il tipo di contenuto che l'utente sta andando ad aggiungere, ovvero un video o un audio.
Per rimuovere un contenuto dai preferiti invece viene utilizzato il metodo `removeFavorite (id:)`, che richiede l'*id* del contenuto da rimuovere. Le funzioni vengono chiamate quando l'utente preme il pulsante per aggiungere o rimuovere un contenuto dai preferiti nella pagina di dettaglio del contenuto.

3.3.3.3 Controllo se un contenuto è tra i preferiti

Quando viene visualizzata la pagina di dettaglio di un contenuto, viene controllato se questo è tra i preferiti dell'utente, tramite il metodo `isFavoriteItem(item:result:)`. Questo metodo richiede un *Item* e ritorna un *Bool*, che indica se il contenuto è tra i preferiti o meno. Il controllo viene effettuato per decidere l'azione che sarà compiuta quando l'utente preme il pulsante per aggiungere o rimuovere un contenuto dai preferiti, e per modificare l'etichetta di tale pulsante.

3.3.4 Gestione dei contenuti in riproduzione

La gestione dei contenuti in riproduzione è fatta da *VideoPlayerInteractor* (listing 3.5), che gestisce le azioni di invio al *backend* dell'attuale tempo di visione e il *fetch* dello stesso per riprendere il contenuto da dove lo si era lasciato.

```
1    protocol VideoPlayerInteractor {
2        func getWatchTime(ofVideo id: String,
3                          result: Binding<Loadable<WatchTime>>)
4        func updateWatchTime(ofVideo id: String,
5                             seconds: Double)
6    }
```

Listing 3.5: Protocollo *VideoPlayerInteractor* per la gestione dei contenuti in riproduzione

3.3.4.1 *Fetch* del tempo di visione

Il *fetch* del tempo di visione è effettuato tramite il metodo `getWatchTime(ofVideo id : result:)`, che richiede l'*id* del contenuto di cui si vuole ottenere il *watch time*, e quest'ultimo viene ritornato in un oggetto. Il metodo viene chiamato quando l'utente avvia la visione di un contenuto, e ritorna un errore 404 in caso l'utente non abbia ancora visionato tale contenuto, altrimenti viene ritornato il tempo di visione e il *video player* viene impostato per riprendere da tale punto.

3.3.4.2 Aggiornamento del tempo di visione

Quando l'utente esce dalla pagina del video player, viene aggiornato il tempo di visione del contenuto. Viene chiamato il metodo `updateWatchTime(ofVideo id: seconds:)`, che richiede l'*id* del contenuto e il tempo di visione attuale in secondi, e aggiorna il tempo di visione del contenuto sul *backend*.

Capitolo 4

Integrazione con Apple Watch e HealthKit

In questo capitolo verrà presentata l'integrazione dell'applicativo *tvOS* con *Apple Watch* e il *framework HealthKit*. In particolare verrà illustrato lo scopo di tale integrazione, lo sviluppo dell'applicativo per *Apple Watch*, le modalità di connessione tra i due applicativi e la raccolta di dati biometrici tramite *HealthKit*.

4.1 Scopo dell'integrazione

Lo scopo della piattaforma sviluppata dall'azienda è permettere ad un utente di visionare video *workout*, per promuovere l'attività fisica e l'allenamento a casa. I video presenti sono realizzati da dei *personal trainers* e sono pensati per essere seguiti come delle lezioni.
Durante questi video, l'azienda ha ritenuto utile che l'utente potesse tenere traccia dei propri dati fisici e biometrici durante l'allenamento, in modo da poterli visualizzare durante il video e poterli anche analizzare in seguito.
Da questo è nata l'idea di integrare l'applicativo *tvOS* con *HealthKit*, ovvero l'*SDK* di *Apple* per la raccolta e l'elaborazione di dati riguardanti la salute e il fitness.
Il modo principale di raccogliere tali dati è utilizzare un *Apple Watch*, ovvero uno smartwatch[G] prodotto da *Apple* che permette di raccogliere dati biometrici tramite dei sensori integrati, come sensori di temperatura, battito cardiaco, accelerometro, giroscopio, ecc.
Apple utilizza i dati raccolti dai sensori per creare in modo automatico delle metriche, come battito cardiaco, calorie bruciate, passi effettuati, movimenti svolti, ecc. Inoltre è possibile, utilizzando *Apple Watch*, avviare un allenamento, in modo da tracciare dati specifici per quel tipo di attività fisica, come ad esempio la corsa, il nuoto, il ciclismo e

molte altre, oltre al tempo di durata di tale allenamento. Gli allenamenti effettuati con questo dispositivo vengono poi salvati automaticamente nell'app *Salute* di *iOS*, comprese le metriche calcolate.
Lo *use case* principale quindi è la possibilità di visualizzare le metriche dell'allenamento in tempo reale, sia su *Apple Watch* sia su *Apple TV*, durante la visione di un video *workout*. Per questo è stata quindi sviluppata un'app per *Apple Watch* che svolge le funzionalità appena descritte.

4.2 Analisi delle metodologie di connessione con Apple Watch

La connessione tra *Apple TV* e *Apple Watch* può essere effettuata in diversi modi, ognuno con le proprie caratteristiche e limitazioni.
Per prima cosa, *Apple Watch* può essere utilizzato per le funzionalità descritte nella sezione 4.1 solo se è presente un *iPhone* associato, in quanto i dati raccolti da *Apple Watch* vengono salvati in automatico nell'app *Salute* di *iOS*.
Per questo motivo, la connessione tra *Apple TV* e *Apple Watch* può essere effettuata in due modi:

- **WatchConnectivity**: questo *framework* permette ad *iPhone* di comunicare con un *Apple Watch* associato ad esso. In questo modo viene effettuata una connessione all'interno della rete locale tra *iPhone* e *Apple TV*, creando un flusso del tipo ***Apple TV*** $\iff$ ***iPhone*** $\iff$ ***Apple Watch***;
- **DeviceDiscoveryUI**: questo *framework* per *tvOS* permette ad *Apple TV* di connettersi direttamente ad un qualsiasi dispostivo *Apple* (in questo caso l'*Apple Watch* dell'utente), senza dover inserire nel mezzo un *iPhone*. La connessione è possibile soltanto se i due dispositivi sono connessi alla stessa rete locale, creando un flusso del tipo ***Apple TV*** $\iff$ ***Apple Watch***.

La scelta tra i due metodi è stata effettuata tenendo conto delle seguenti considerazioni:

- La connessione deve essere il più possibile semplice e trasparente per l'utente, in modo da non dover effettuare operazioni complicate per poter utilizzare l'applicativo;
- La connessione deve essere il più possibile veloce, in modo da poter visualizzare i dati in tempo reale durante la visione del video *workout*.

È stato quindi scelto di utilizzare il *framework DeviceDiscoveryUI*, che rende la connessione più veloce e semplice senza aggiungere operazioni aggiuntive per l'utente o dispositivi aggiuntivi.

4.3 Sviluppo dell'applicativo per Apple Watch

Questa sezione descrive come è stata implementata l'interfaccia dell'applicativo, oltre alla parte di *business logic*, indicando le scelte di design fatte e le motivazioni che hanno portato a tali scelte.

4.3.1 Linee guida

Da parte dell'azienda, non sono stati creati dei *mockup* per l'applicativo, e non sono stati definiti dei requisiti, quindi mi sono basato sulle linee guida di *Apple* per lo sviluppo di applicativi per *Apple Watch* (fonti: [5] [14]) e ho definito autonomamente dei requisiti, ovvero:

- L'applicativo deve essere semplice e intuitivo da utilizzare;
- L'applicativo deve registrare i dati biometrici dell'utente durante la visione di un video *workout*, tramite l'utilizzo degli allenamenti di *Apple Watch*;
- Deve essere possibile interfacciarsi con *Apple TV* per condividere i dati biometrici registrati e visualizzarli sullo schermo del televisore.

Le linee guida consigliano fortemente che un'applicazione per *Apple Watch* che è utilizzata per registrare degli allenamenti debba avere una struttura standard per le viste che visualizzano l'allenamento in corso:

- **Pagina delle metriche**: questa è la pagina principale delle viste che visualizzano un allenamento in corso. Qui vengono visualizzati i dati relativi a tale allenamento, come il tempo trascorso dall'inizio e i principali dati biometrici, come il battito cardiaco, le calorie bruciate, la distanza percorsa (se applicabile), ecc;
- **Pagina dei controlli**: questa pagina permette di interagire con l'allenamento in corso tramite una serie di pulsanti che svolgono delle azioni su tale allenamento, come per esempio metterlo in pausa, riprenderlo, terminarlo, ecc;
- **Pagina *Now Playing***: questa pagina permette di visualizzare, se presente, i controlli per la gestione della musica in riproduzione su *Apple Watch*, indipendentemente dall'applicativo che sta riproducendo la musica. Questa pagina è facoltativa, e nel caso dell'applicazione sviluppata non è stata implementata, in quanto sono stati inseriti dei controlli custom per la gestione del video *workout* in riproduzione su *Apple TV*.

Queste viste devono essere organizzate tramite un'interfaccia a schede, che permette di scorrere orizzontalmente tra le tre pagine. L'ordine delle pagine è standard e deve essere rispettato, ovvero: *Pagina dei controlli* — *Pagina delle metriche* — *Pagina* Now Playing.

4.3.2 Struttura dell'applicativo

In base alle linee guida appena descritte, e ai requisiti definiti precedentemente, l'applicativo è stato strutturato nel seguente modo:

- Pagina di connessione
- Pagina di inizio allenamento
- Pagina dell'allenamento
- Pagina di riepilogo dell'allenamento

4.3.2.1 Pagina di connessione

Questa pagina viene visualizzata quando l'applicazione è in attesa dell'avvio della connessione da parte di *Apple TV*. In questa pagina viene visualizzato un messaggio di attesa e un'animazione di caricamento (figura 4.1).

Figura 4.1: Apple Watch — Pagina di connessione

4.3.2.2 Pagina di inizio allenamento

Questa pagina visualizza le caratteristiche del *workout* che sta per essere avviato su *Apple TV*: vengono visualizzati il titolo, la durata in minuti dell'allenamento e anche un pulsante che permette di avviare l'allenamento contemporaneamente sia su *Apple Watch* che su *Apple TV* (figura 4.2).

Figura 4.2: Apple Watch — Pagina di inizio allenamento

4.3.2.3 Pagina dell'allenamento

Questa pagina visualizza i dati relativi all'allenamento in corso, ed è strutturata come descritto nella sezione 4.3.1, ovvero è una vista a schede che permette di scorrere orizzontalmente tra le due pagine che la compongono: metriche e controlli.

Pagina delle metriche In questa pagina vengono visualizzati i dati relativi all'allenamento in corso, nello specifico il tempo trascorso dall'inizio dell'allenamento, il battito cardiaco, e le calorie bruciate (figura 4.3).

Figura 4.3: Apple Watch — Pagina delle metriche di allenamento

Pagina dei controlli In questa pagina (figura 4.4) vengono visualizzate le azioni tramite le quali è possibile interagire con l'allenamento in corso, nel dettaglio è possibile

metterlo in pausa (automaticamente viene messo in pausa anche il video su *Apple TV*) o terminarlo (automaticamente viene terminato anche il video su *Apple TV*).

Figura 4.4: Apple Watch — Pagina dei controlli dell'allenamento

4.3.2.4 Pagina di riepilogo dell'allenamento

Questa pagina viene visualizzata quando l'allenamento è terminato, e visualizza un riepilogo del *workout*, con il tempo totale trascorso, il battito cardiaco medio, le calorie bruciate totali e gli *Activity Rings* di *Apple Watch* aggiornati con i dati dell'allenamento appena terminato (figura 4.5).

Figura 4.5: Apple Watch — Pagina di riepilogo dell'allenamento

4.3.3 Raccolta di dati biometrici tramite HealthKit

La raccolta dei dati biometrici è stata implementata tramite *HealthKit*, un *framework* che permette di accedere ai dati biometrici raccolti da *Apple Watch* e dati sulla salute

che l'utente ha inserito manualmente o che sono stati estrapolati dal sistema.
La registrazione dei dati è stata implementata tramite l'avvio di un allenamento di *Apple Watch*. Gli allenamenti sono una funzionalità nativa dell'orologio che permette di registrare un allenamento di un certo tipo, come per esempio una corsa, una camminata, una nuotata, ecc. . . , e di tenere traccia dei propri dati biometrici durante tale attività. *Apple Watch* fornisce di default molti tipi di allenamenti, però per l'app da me sviluppata viene utilizzato un allenamento di tipo generico, ovvero non associato ad alcun tipo di attività, in quanto i video *workout* che vengono riprodotti su *Apple TV* possono essere di tipi diversi (e non necessariamente combaciano con i tipi di allenamenti offerti da *Apple Watch*).

Il flusso di raccolta dei dati biometrici è il seguente:

- Richiesta di autorizzazione per l'accesso ai dati di *HealthKit* e al sistema di allenamento di *Apple Watch*;
- Avvio dell'allenamento e visualizzazione dei dati biometrici su *Apple Watch* (e invio ad *Apple TV*);
- Termine dell'allenamento e visualizzazione del riepilogo su *Apple Watch*.

4.3.3.1 Richiesta di autorizzazione per l'accesso ai dati di HealthKit

Apple non permette alle app di accedere ai dati personali dell'utente se esso non esprime esplicitamente il proprio consenso, in quanto questi potrebbero contenere dati sensibili che l'utente non vuole condividere. Per questo motivo, per accedere al sistema di allenamento e ai dati salvati in *HealthKit*, è necessario richiedere all'utente il consenso all'accesso ai dati.
Le classi utilizzate per accedere ad *HealthKit* forniscono già un metodo di default per effettuare questa richiesta, che si occupa di mostrare all'utente un *popup* di richiesta di autorizzazione al primo avvio dell'app, e di gestire la risposta dell'utente.
Il metodo richiede che venga fornita una lista dei dati a cui si vuole accedere, in modo da evitare un accesso incondizionato a tutti i dati sanitari. Il *popup* permette inoltre all'utente di scegliere quali dati della lista condividere e quali no. Per l'applicazione sviluppata, i dati richiesti sono i seguenti:

- Battito cardiaco
- Calorie bruciate
- Riepilogo dell'allenamento

Una volta che l'utente ha concesso l'autorizzazione, l'app può accedere ai dati richiesti, anche in futuro, senza dover richiedere nuovamente il consenso.

4.3.3.2 Avvio dell'allenamento e visualizzazione dei dati biometrici

Una volta fornito il consenso, viene visualizzata la schermata di inizio allenamento (se l'app è connessa ad *Apple TV*), e l'utente può avviare l'allenamento.
Una volta avviato, il *workout* inizia ad essere registrato da *Apple Watch*, e i dati biometrici vengono visualizzati nella schermata delle metriche. L'allenamento continua ad essere registrato anche se l'app viene messa in *background*, per esempio quando lo schermo va in *timeout* (lo schermo si spegne dopo non essere stato utilizzato per un certo periodo) o quando l'utente gira il polso (azione che provoca lo spegnimento dello schermo).
Durante la registrazione del *workout*, al cambiamento dei dati biometrici questi vengono inviati all'app su *Apple TV*, che visualizza tali dati in un *overlay* sopra al video, in modo da non interrompere o interferire con la riproduzione dello stesso.

4.3.3.3 Terminazione dell'allenamento e visualizzazione del riepilogo

Quando l'utente termina l'allenamento, *Apple Watch* lo registra automaticamente all'interno dell'app *Fitness* su *iPhone*, che mantiene un database di tutti gli allenamenti effettuati e dei dati registrati durante gli stessi. Inoltre sull'orologio viene visualizzata una schermata di riepilogo che visualizza i dati del *workout* appena terminato. Questi dati sono metriche calcolate automaticamente da *Apple Watch* e interpolate dai dati registrati. Nel caso dell'app sviluppata, i dati visualizzati nel riepilogo sono i seguenti:

- Tempo totale di allenamento
- Battito cardiaco medio
- Calorie bruciate totali
- *Activity Rings* aggiornati

Gli *Activity Rings* (figura 4.6) sono tre anelli che rappresentano tre obiettivi giornalieri di attività fisica, e che vengono aggiornati automaticamente da *Apple Watch* quando vengono registrati degli allenamenti. Sono utilizzati per tenere traccia dell'attività fisica giornaliera dell'utente e sono un tentativo di gamification[G] di tale attività, in quanto è presente anche un sistema di medaglie che vengono sbloccate al raggiungimento degli obiettivi. Gli anelli sono tre e rappresentano tre diversi obiettivi:

- **Movimento**: indica il numero di calorie bruciate durante l'attività fisica nell'arco della giornata;
- **Esercizio**: indica il numero di minuti di attività fisica effettuati durante la giornata;
- **In piedi**: indica il numero di ore in cui l'utente si è alzato in piedi durante la giornata.

Figura 4.6: Apple Watch — *Activity Rings*

Il livello giornaliero per ognuno dei tre obiettivi è scelto dall'utente, e gli allenamenti registrati dall'app sviluppata contribuiscono al raggiungimento di questi obiettivi, in quanto vengono utilizzati i sistemi di allenamento e di raccolta dati nativi ad *Apple Watch*.

4.4 Integrazione della connessione nell'applicativo tvOS

Per completare l'integrazione, è stato necessario effettuare delle modifiche all'applicativo *tvOS*, in modo da utilizzare il *framework DeviceDiscoveryUI* e gestire la visualizzazione dei dati biometrici sullo schermo del televisore. Le modifiche apportate sono state le seguenti:

- Aggiunta dell'interfaccia di avvio della connessione;
- Aggiunta del codice di gestione della connessione e scambio di dati;
- Aggiunta dell'interfaccia di visualizzazione dei dati biometrici.

4.4.1 Interfaccia di avvio della connessione

Per permettere un accesso veloce da parte dell'utente alla funzionalità di connessione con l'orologio, ho modificato la pagina di dettaglio del contenuto, inserendo un pulsante che permette di aprire l'interfaccia di avvio della connessione (figura 4.7).

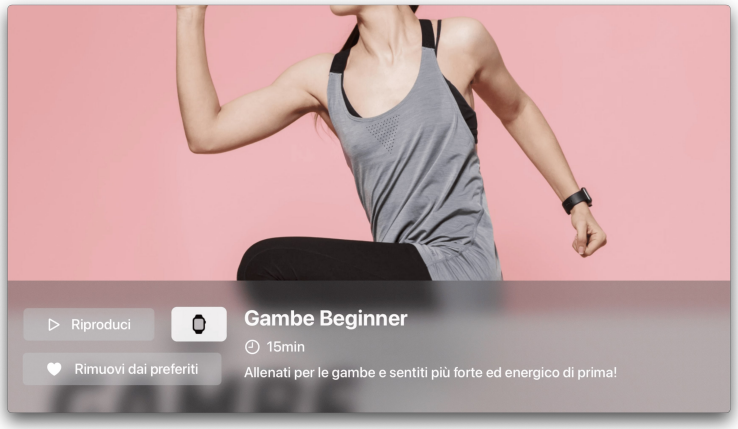


Figura 4.7: Pagina di dettaglio del contenuto con pulsante per la connessione con Apple Watch

L'interfaccia è gestita interamente dal *framework DeviceDiscoveryUI*, e permette di visualizzare la lista dei dispositivi *Apple Watch* connessi alla stessa rete locale, collegati allo stesso *account Apple*, e di selezionare l'orologio con cui si vuole connettersi (figura 4.8, fonte: [1]). Viene anche visualizzato un messaggio di istruzioni e una descrizione dello scopo della connessione con l'orologio. Una volta selezionato, sull'orologio viene visualizzato un messaggio che chiede di confermare la connessione con *Apple TV* (figura 4.8), e una volta confermata, la connessione viene avviata ed è possibile iniziare a scambiare dati.

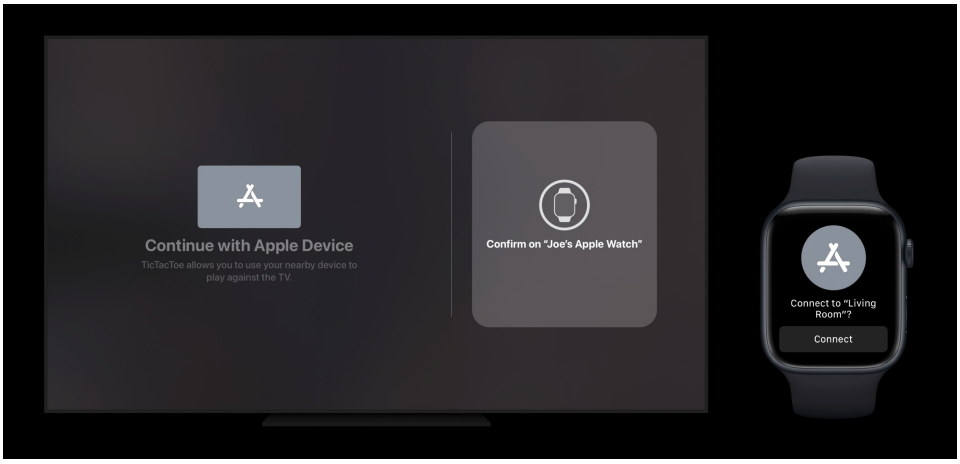


Figura 4.8: Interfaccia di *DeviceDiscoveryUI* e di conferma della connessione su Apple Watch

4.4.2 Codice di gestione della connessione e scambio di dati

Il codice della gestione della connessione dello scambio dei dati è completamente custom per l'applicazione.
Per prima cosa una volta accettato il collegamento sull'orologio, *Apple TV* avvia effettivamente la connessione, creando un oggetto che rappresenta la connessione stessa e che permette di inviare e ricevere dati. Allo stesso momento sull'orologio viene creato un oggetto che si mette in ascolto per richieste di connessione da parte della tv.
Una volta aperto il collegamento con successo, è possibile avviare lo scambio di dati. Lo scambio è fatto a livello di *byte*, quindi all'invio i dati devono essere convertiti in *byte*, e all'arrivo devono essere riconvertiti in oggetti utilizzabili dall'applicazione.
Da parte dell'orologio, ogni volta che i dati biometrici subiscono una modifica, i nuovi valori sono inviati alla tv.
Da parte della tv, ogni volta che i *byte* sono ricevuti e sono stati convertiti in oggetti, vengono aggiornati i valori visualizzati sullo schermo.

La connessione tra orologio e tv non è solo utilizzata per lo scambio di dati biometrici, ma anche per la gestione della riproduzione dell'allenamento sul televisore. Infatti, quando l'utente avvia l'allenamento sull'orologio, viene inviato un messaggio alla tv che avvia la riproduzione del video *workout*. In aggiunta, è possibile anche mettere in pausa l'allenamento dall'orologio e terminarlo. Queste azioni possono inoltre essere svolte sulla tv ed avranno effetto anche sull'allenamento sull'orologio.

4.4.2.1 Interfaccia di visualizzazione dei dati biometrici

Per la visualizzazione dei dati biometrici sul televisore ho deciso di utilizzare un *overlay* sopra al video dell'allenamento (figura 4.9), in modo da poterli tenere costantemente visibili ma in modo che non distraessero l'utente dal contenuto in riproduzione.
L'*overlay* è visualizzato nell'angolo in alto a sinistra dello schermo del televisore, e mostra il battito cardiaco e le calorie bruciate in tempo reale.
Inoltre l'*overlay* viene visualizzato solamente se è stato connesso un *Apple Watch*, in modo da renderlo nascosto per gli utenti che non vogliono utilizzare tale funzionalità.

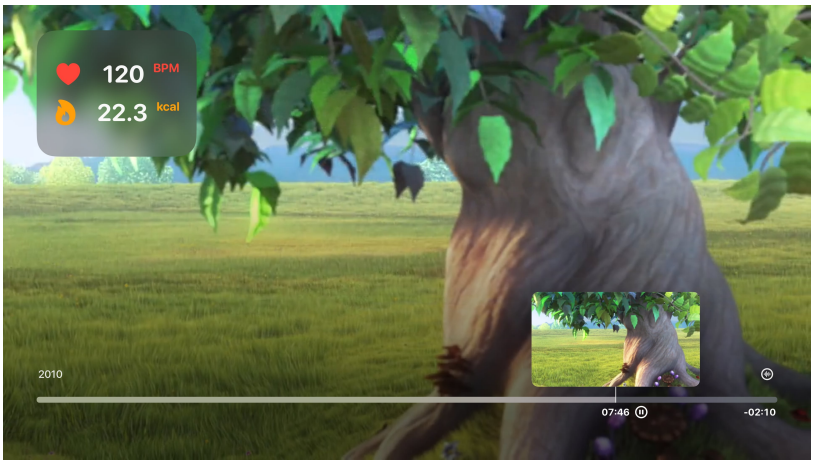


Figura 4.9: Overlay per i dati biometrici nella schermata del video player

4.5 Problemi riscontrati

Durante lo sviluppo dell'integrazione con *Apple Watch* e *HealthKit* è stato riscontrato un problema, che non è stato possibile risolvere.
Il problema in questione riguarda il *framework DeviceDiscoveryUI* nativo a *tvOS*, fondamentale per poter effettuare una ricerca sulla rete locale degli *Apple Watch*, e per la successiva apertura della connessione tra i due dispositivi. Il problema risiede proprio nell'apertura della connessione.
Durante la visualizzazione dell'interfaccia di ricerca dei dispositivi, gli *Apple Watch* sulla rete locale e collegati allo stesso *account Apple* del televisore vengono visualizzati correttamente. Inoltre, una volta selezionato l'orologio con cui si vuole connettersi, il messaggio di conferma sull'orologio viene anch'esso visualizzato correttamente.
Quando però il *framework* dovrebbe iniziare la connessione, il processo si ferma, e la conferma inviata dall'orologio viene ignorata da *Apple TV*.
Un primo tentativo di risoluzione è stato quello di utilizzare una connessione *WiFi* diversa e privata tra i due dispositivi, in quanto la rete aziendale non permette la connessione su alcune porte, però il problema è rimasto.
Successivamente ho riscontrato che durante l'esecuzione dell'applicativo *tvOS* in modalità di *debug*, ovvero connesso al *Mac*, sulla *console* dell'IDE viene visualizzato un messaggio di errore che indicherebbe la mancanza di alcuni certificati. Dopo molte ricerche insieme agli altri sviluppatori *iOS* dell'azienda, si è concluso che i certificati in questione sono proprietari ad *Apple* e non è possibile installarli manualmente, e che quindi non fossero la causa del problema di connessione. Per verificare ulteriormente che i certificati non fossero un problema, ho provato ad eseguire una compilazione dell'app con il mio account sviluppatore personale, che tramite *Xcode* scarica automaticamente tutti i certificati necessari per lo sviluppo. In questo modo ho potuto confermare che i

certificati non fossero la causa dell'impossibilità di avviare la connessione, in quanto tutti quelli necessari al funzionamento erano stati correttamente installati. La conclusione quindi è stata che gli errori visualizzati fossero fuorvianti e non rappresentassero la causa del problema (la visualizzazione di errori non rilevanti ad un problema è una problematica nota di *Xcode*).
L'unica conclusione, accettata anche dal tutor interno e dal responsabile dello sviluppo mobile, è stata quindi che il *framework DeviceDiscoveryUI* presenta un *bug* che non permette l'apertura della connessione tra due dispositivi. Per questo motivo non è stato possibile completare la funzionalità di connessione e scambio dati tra *Apple TV* e *Apple Watch*, sebbene il codice per la comunicazione fosse già stato implementato.

Capitolo 5

Testing e Collaudo

In questo capitolo sono presentate le modalità di test e collaudo degli applicativi sviluppati, in particolare sono illustrate le tecniche usate, come l'utilizzo di simulatori durante lo sviluppo e anche l'esecuzione degli applicativi su dispositivi fisici.

5.1 Versioni di applicativi e sistemi operativi utilizzati

Durante lo sviluppo sono stati utilizzati i seguenti strumenti:

- **Computer adibito allo sviluppo**: MacBook Air M2
- **Sistema operativo**: macOS Ventura 13.4.1 (c)
- **IDE**: Xcode 13.4.1
- **Linguaggio di programmazione**: Swift 5.8.1
- **Simulatore 1**: Apple TV 4K (terza generazione) — tvOS 16.4
- **Simulatore 2**: Apple Watch Series 8 (45mm) — watchOS 9.4
- **Dispositivo 1**: Apple TV 4K (terza generazione, WiFi + Ethernet) — tvOS 16.5 - 16.6
- **Dispositivo 2**: Apple Watch Series 6 (GPS + Cellular) — watchOS 9.5

5.2 Tipologie di test

5.2.1 Test di unità

Di seguito vengono riportati i test di unità effettuati per le varie componenti del progetto. Questi test sono stati eseguiti per verificare il corretto funzionamento e

rendering[G] delle singole componenti.
I test di unità sono identificati da un codice univoco, secondo la seguente convenzione:

[TU][Numero]

dove:

- **TU**: abbreviazione di *Test di Unità*;
- **Numero**: indica il numero del test.

L'esito del test può assumere uno dei seguenti due valori:

- **Superato**: se il test è stato completato con successo;
- **Non superato**: se il test non è stato completato con successo.

5.2.1.1 Test di unità dell'applicazione *tvOS*

Tabella 5.1: Test di unità dell'applicazione tvOS

Codice	Componente	Descrizione	Esito
TU1	VideoButton	Verifica del corretto *rendering* del pulsante che identifica un contenuto, con la visualizzazione del titolo e della *thumbnail*	Superato
TU2	HeroVideoButton	Verifica del corretto *rendering* del pulsante che identifica un contenuto nel carosello *Hero*, con la visualizzazione del titolo e della *thumbnail*	Superato
TU3	CustomDevicePicker	Verifica del corretto *rendering* del componente che permette di selezionare un dispositivo *Apple* nella rete locale a cui connettersi	Superato
TU4	QRCodeGenerator	Verifica della corretta generazione di un *QRCode* data una stringa	Superato

5.2.1.2 Test di unità dell'applicazione *watchOS*

Tabella 5.2: Test di unità dell'applicazione watchOS

Codice	Componente	Descrizione	Esito
TU5	ElapsedTimeView	Verifica del corretto *rendering* del componente che identifica la durata del *workout* in corso	Superato
TU6	MetricsView	Verifica del corretto *rendering* del componente che identifica le metriche del *workout* in corso	Superato
TU7	ControlsView	Verifica del corretto *rendering* del componente che identifica i controlli per la gestione del *workout* in corso	Superato
TU8	ActivityRingsView	Verifica del corretto *rendering* del componente che identifica gli *Activity Rings*	Superato
TU9	AwaitingConnectionView	Verifica del corretto *rendering* del componente che identifica la schermata di attesa della connessione con *Apple TV*	Superato
TU10	Haptics	Verifica del corretto funzionamento dei feedback aptici[G] al tap su un pulsante	Superato

5.2.2 Test di integrazione

Di seguito vengono riportati i test di integrazione effettuati per le varie componenti del progetto. Questi test sono stati eseguiti per testare il corretto funzionamento delle interazioni tra le varie componenti.
I test di integrazione sono identificati da un codice univoco, secondo la seguente convenzione:

[TI][Numero]

dove:

- **TI**: abbreviazione di *Test di Integrazione*;
- **Numero**: indica il numero del test.

L'esito del test può assumere uno dei seguenti due valori:

- **Superato**: se il test è stato completato con successo;
- **Non superato**: se il test non è stato completato con successo.

5.2.2.1 Test di integrazione dell'applicazione *tvOS*

Tabella 5.3: Test di integrazione dell'applicazione tvOS

Codice	Componente	Descrizione	Esito
TI1	HeroCarousel	Verifica del corretto *rendering* del carosello *Hero*, formato da una serie di *HeroVideoButton*, e verifica delle interazioni con esso tramite telecomando	Superato
TI2	ComponentView	Verifica del corretto *rendering* del carosello orizzontale in ogni pagina in cui è usato, e del *rendering* dei *VideoButton* di cui è formato	Superato
TI3	FavoriteView	Verifica del corretto *rendering* della pagina *FavoriteView*, formata da due caroselli di elementi preferiti (*ComponentView*)	Superato
TI4	MainView	Verifica del corretto *rendering* della vista principale, che agisce da *wrapper* per le pagine dell'app e permette la navigazione per schede	Superato
TI5	PageView	Verifica del corretto *rendering* delle singole *PageView*, che identificano le tre pagine principali (Home, Musica, Generi)	Superato
TI6	ProfileView	Verifica del corretto *rendering* della pagina del profilo	Superato
TI7	QRLoginView	Verifica del corretto *rendering* della pagina di login con *QRCode*, e verifica della corretta generazione del *QRCode* e link per effettuare il login	Superato
TI8	SearchView	Verifica del corretto *rendering* della pagina di ricerca e del funzionamento delle funzionalità di ricerca	Superato
TI9	SplashView	Verifica del corretto *rendering* della pagina *splash*, visualizzata durante l'avvio dell'app	Superato

TI10	VideoView	Verifica del corretto *rendering* della pagina di dettaglio del contenuto e del corretto *fetch* dei dati del contenuto	Superato
TI11	VideoPlayerView	Verifica del corretto *rendering* della pagina di visualizzazione del contenuto, e verifica del funzionamento del *video player* e delle funzionalità che permettono di riprendere un contenuto dalla visualizzazione precedente	Superato

5.2.2.2 Test di integrazione dell'applicazione *watchOS*

Tabella 5.4: Test di integrazione dell'applicazione watchOS

Codice	Componente	Descrizione	Esito
TI12	ContentView	Verifica del corretto *rendering* della pagina principale e funzionamento del sistema di *routing*	Superato
TI13	WorkoutView	Verifica del corretto *rendering* della pagina di allenamento, con visualizzazione dei controlli e delle metriche	Superato
TI14	SummaryView	Visualizzazione del corretto *rendering* della pagina di riepilogo dell'allenamento, con le metriche calcolate correttamente al termine dell'allenamento	Superato

5.2.3 Test di regressione

Di seguito vengono riportati i test di regressione effettuati per le varie componenti del progetto. Questi test sono stati eseguiti per testare il corretto funzionamento delle componenti dopo l'implementazione di nuove funzionalità.
I test di regressione sono identificati da un codice univoco, secondo la seguente convenzione:

[TR][Numero]

dove:

- **TR**: abbreviazione di *Test di Regressione*;
- **Numero**: indica il numero del test.

L'esito del test può assumere uno dei seguenti due valori:

- **Superato**: se il test è stato completato con successo;
- **Non superato**: se il test non è stato completato con successo.

5.2.3.1 Test di regressione dell'applicazione *tvOS*

Tabella 5.5: Test di regressione dell'applicazione tvOS

Codice	Componente	Descrizione	Esito
TR1	MainView	Viene testato il corretto *rendering* e funzionamento della vista dopo l'aggiunta del carosello *Hero*	Superato
TR2	PageView	Viene testato il corretto *rendering* e funzionamento della vista dopo l'aggiunta del carosello *Hero*	Superato
TR3	MainView	Viene testato il corretto *rendering* e funzionamento della vista dopo le modifiche al design di *VideoButton*	Superato
TR4	PageView	Viene testato il corretto *rendering* e funzionamento della vista dopo le modifiche al design di *VideoButton*	Superato
TR5	SearchView	Viene testato il corretto *rendering* e funzionamento della vista dopo le modifiche al design di *VideoButton*	Superato
TR6	FavoriteView	Viene testato il corretto *rendering* e funzionamento della vista dopo le modifiche al design di *VideoButton*	Superato
TR7	ComponentView	Viene testato il corretto *rendering* e funzionamento della vista dopo le modifiche al design di *VideoButton*	Superato

5.2.3.2 Test di regressione dell'applicazione *watchOS*

Tabella 5.6: Test di regressione dell'applicazione watchOS

Codice	Componente	Descrizione	Esito
TR8	WorkoutView	Viene testato il corretto *rendering* e funzionamento della vista dopo l'aggiunta di *ElapsedTimeView*	Superato
TR9	WorkoutView	Viene testato il corretto *rendering* e funzionamento della vista dopo l'aggiunta dei *feedback aptici*	Superato
TR10	ContentView	Viene testato il corretto *rendering* e funzionamento della vista dopo l'aggiunta di *SummaryView*	Superato
TR11	SummaryView	Viene testato il corretto *rendering* e funzionamento della vista dopo l'aggiunta di *ActivityRingsView*	Superato

5.3 Utilizzo dei simulatori

Durante lo sviluppo degli applicativi, per effettuare test i test di funzionamento sono stati usati i simulatori di *Apple TV* e *Apple Watch* forniti da *Xcode*. Questi simulatori permettono di testare le funzionalità degli applicativi senza dover utilizzare dispositivi fisici.

5.3.1 Simulatore di *Apple TV*

Il simulatore di *Apple TV* (figura 5.1) permette di eseguire localmente una versione di *tvOS* su un computer *Mac*. Il simulatore dispone di tutte le funzionalità principali di *tvOS*, compreso anche un telecomando virtuale per testare le interazioni con esso. Per la maggior parte dello sviluppo è stato utilizzato il simulatore, questo però non offre la possibilità di testare le funzionalità di connessione con altri dispositivi *Apple*, necessaria per testare la comunicazione tra *Apple TV* e *Apple Watch* e la condivisione dei dati biometrici tramite *DeviceDiscoveryUI*.

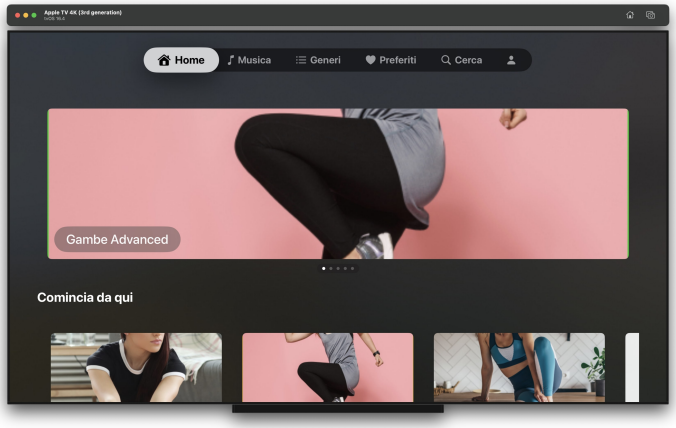


Figura 5.1: Simulatore Apple TV in esecuzione su Mac

5.3.2 Simulatore di *Apple Watch*

Il simulatore di *Apple Watch* (figura 5.2) permette di eseguire localmente una versione di *watchOS* su un computer *Mac*. Tale simulatore offre tutte le funzionalità principali di *Apple Watch*, compresi anche i tasti fisici come la *Digital Crown* (la rotella presente sul lato di *Apple Watch*), indispensabile per testare l'interazione con l'interfaccia grafica dell'applicazione. Anche in questo caso il simulatore non permette di testare le funzionalità di connessione offerte dal *framework DeviceDiscoveryUI*.

Figura 5.2: Simulatore Apple Watch in esecuzione su Mac

5.4 Utilizzo dei dispositivi fisici

Per effettuare dei test più approfonditi e verificare il funzionamento degli applicativi in condizioni reali di utilizzo, ho avuto la possibilità di utilizzare anche dei dispositivi fisici,

forniti dall'azienda. Questi dispositivi sono stati fondamentali per ultimare lo sviluppo in quanto hanno permesso di testare le app sui dispositivi che effettivamente verranno utilizzati dagli utenti finali, e questo ha dato modo di riscontrare alcuni problemi che non erano emersi durante lo sviluppo con i simulatori, come alcuni problemi di *layout* e dimensioni dei caratteri, soprattutto per quanto riguarda l'app per *Apple TV*, che viene visualizzata su schermi di televisori di grandi dimensioni.

5.4.1 *Apple TV*

Il dispositivo *Apple TV* fornito dall'azienda è stata un *Apple TV 4K* di terza generazione. Questo mi ha permesso di testare l'applicativo su diversi *range* di risoluzioni, dal *Full HD* al *4K*, e di verificare il corretto funzionamento dell'applicativo su schermi di grandi dimensioni. Usare tale dispositivo ha inoltre permesso di testare la connessione con *Apple Watch* tramite il *framework DeviceDiscoveryUI*, purtroppo però a causa di un problema legato ad un *bug* del *framework* installato sul dispositivo, la connessione non è mai stata stabilita correttamente.

5.4.2 *Apple Watch*

Il dispositivo *Apple Watch* fornito dall'azienda è stato un *Apple Watch Series 6*, non un dispositivo di ultima generazione come *Apple TV*, però questo non ha causato problemi durante lo sviluppo, ha permesso invece di testare il corretto funzionamento dell'applicativo anche su dispositivi meno recenti e con dimensioni dello schermo più piccole.

5.5 Collaudo e accettazione

Durante lo sviluppo del progetto, sono stati effettuati degli incontri con il tutor aziendale e il responsabile dello sviluppo *mobile* dell'azienda, al fine di verificare lo stato di avanzamento del progetto e di valutare le funzionalità implementate.
Inoltre è stato svolto anche un incontro finale con il responsabile dello sviluppo *mobile* per valutare gli applicativi sviluppati e effettuare una dimostrazione di funzionamento.

Capitolo 6

Conclusioni

L'obiettivo dell'attività di stage svolta presso Rawfish è stato quello di realizzare un'app per dispositivi *Apple TV* basata sulla piattaforma interna di distribuzione e visualizzazione di contenuti audio e video legati al *fitness*.
Nello specifico, l'applicazione doveva permettere all'utente di accedere alla piattaforma, navigarne i contenuti, permettere di riprodurli e gestire la propria lista di preferiti, il tutto tramite un'interfaccia utente semplice e intuitiva, navigabile su un televisore tramite l'utilizzo di un telecomando con limitate possibilità di interazione.
Infine è stato richiesto di esplorare e implementare la possibilità di tenere traccia in tempo reale dei dati biometrici dell'utente durante lo svolgimento di un video *workout*, tramite l'utilizzo di *Apple Watch*.
Per poter realizzare tutto ciò è stata necessaria una fase iniziale di studio delle tecnologie e tecniche di sviluppo per piattaforme *Apple*, in particolare il linguaggio *Swift* e il *framework SwiftUI*.

Per quanto riguarda gli obiettivi prefissati, sono stati raggiunti tutti gli obiettivi obbligatori e facoltativi, mentre degli obiettivi desiderabili solo uno non è stato raggiunto, in quanto non è stato possibile implementare la comunicazione tra *Apple TV* e *Apple Watch*.

Più nello specifico, sono stati raggiunti i seguenti obiettivi obbligatori:

- **O1**: dimestichezza con l'*IDE* di sviluppo per applicazioni *iOS Xcode*;
- **O2**: sviluppo di un applicativo in *Swift*;
- **O3**: implementazione interazione con strutture server;
- **O4**: interazione con *tvOS*.

Sono stati raggiunti anche i seguenti obiettivi desiderabili:

- **D1**: sviluppo di interfacce complesse;
- **D2**: gestione *AVPlayer* (*SDK streaming* nativo);
- **D3**: proattività sul progetto;

Come indicato precedentemente, il requisito **D4** non è stato raggiunto, in quanto non è stato possibile implementare la comunicazione tra *Apple TV* e *Apple Watch*, nonostante la raccolta dei dati da parte di quest'ultimo e l'integrazione con *HealthKit* e il sistema nativo di registrazione degli allenamenti fosse già stata implementata. Il requisito non è stato raggiunto in quanto il *framework* scelto per la comunicazione tra i due dispositivi, *DeviceDiscoveryUI*, non ha permesso di stabilire in modo corretto la connessione tra i due dispositivi, probabilmente a causa di un bug del *framework* stesso.

Infine, è stato raggiunto anche l'obiettivo facoltativo:

- **F1**: integrazione con strumenti e flusso di lavoro aziendale.

Questo obiettivo indica come io abbia avuto modo di integrarmi con il team di sviluppo dell'azienda, andando poi a utilizzare gli strumenti e le tecniche di sviluppo da loro stabilite. Più nello specifico, ho utilizzato una *repository* aziendale su *GitLab* per il versionamento del codice, e *Google Chat* per le comunicazioni con il team. Inoltre ho avuto accesso ad alcune *repository* interne relative ad altri progetti, per poter studiare il codice e le tecniche di sviluppo aziendali.

6.1 Conoscenze e competenze acquisite

Durante il mio periodo di stage, ho avuto l'opportunità di acquisire una vasta gamma di conoscenze e competenze nel campo dello sviluppo di applicazioni per piattaforme *Apple*. Una delle competenze più significative è stata l'approfondita familiarità con *SwiftUI*, con il quale ho sviluppato le interfacce grafiche per le app. Ho imparato a utilizzare le funzionalità del *framework* per creare *layout* dinamici e reattivi, implementando interfacce complesse e intuitive. Questa competenza mi ha consentito di fornire agli utenti un'esperienza coinvolgente e di alta qualità, favorendo l'accessibilità e l'usabilità dell'applicazione.

L'apprendimento del linguaggio di programmazione *Swift* è stato cruciale per imparare a sviluppare per tutte le piattaforme *Apple*. Durante lo stage, ho affinato le mie competenze nella scrittura di codice *Swift*, utilizzando efficacemente le caratteristiche del linguaggio per creare una logica di funzionamento solida e efficiente. Inoltre, ho sviluppato una padronanza delle convenzioni di programmazione e delle *best practice*, garantendo una struttura modulare e manutenibile dell'applicazione.

Un altro traguardo importante è stata la gestione dei contenuti multimediali della piattaforma. Ho imparato a gestire il caricamento e la riproduzione di contenuti video e audio, gestendo inoltre anche il riavvio della visione del contenuto dal punto dove era terminata la riproduzione precedente.

L'implementazione dell'integrazione con dispositivi *Apple*, come *Apple Watch* e *HealthKit*, ha ampliato ulteriormente le mie competenze. Ho appreso come avviare una comunicazione tra due dispositivi, acquisendo una comprensione dei protocolli di comunicazione e delle interfacce di programmazione. Questo mi ha permesso di arricchire l'applicazione con funzionalità di condivisione dei dati biometrici, migliorando l'esperienza dell'utente. Ho inoltre imparato a interfacciarmi con i servizi di *HealthKit* per la raccolta e la gestione dei dati sanitari e fisici, utilizzando le funzionalità del *framework* per fornire all'utente un'esperienza di monitoraggio completa e accurata.

Inoltre, ho sviluppato competenze di collaborazione e comunicazione all'interno di un team di sviluppo. Ho imparato a condividere idee, affrontare sfide tecniche e adattarmi alle dinamiche lavorative aziendali.

6.2 Valutazione personale

L'esperienza di stage presso Rawfish è stata senza dubbio un capitolo formativo e arricchente nel mio percorso professionale. Durante questo periodo, ho avuto l'opportunità di immergermi nel mondo dello sviluppo di applicazioni *iOS* e *tvOS* in un contesto aziendale dinamico e all'avanguardia. L'azienda si è dimostrata un ambiente stimolante in cui ho potuto mettere in pratica le conoscenze teoriche acquisite durante gli studi e apprendere nuove competenze in modo pratico ed efficace.

Uno dei punti salienti di questa esperienza è stata l'approfondita formazione tecnica ricevuta. Sono grato al tutor aziendale per la guida preziosa e le conoscenze condivise durante tutto il processo. Questo mi ha permesso di apprendere le *best practice* nello sviluppo di applicazioni, dal *design* delle interfacce utente all'implementazione di funzionalità complesse. L'opportunità di lavorare su una piattaforma *OTT* e di contribuire alla creazione di un'applicazione per la distribuzione di contenuti audio e video mi ha fornito una visione concreta e pratica del ciclo di sviluppo *software*.

Ho apprezzato particolarmente la varietà di obiettivi assegnati durante lo stage, che mi hanno consentito di adattare la mia esperienza alle aspirazioni e alle sfide che desideravo affrontare. La flessibilità nell'approccio agli obiettivi ha sottolineato l'attenzione dell'azienda nel promuovere la crescita individuale e il coinvolgimento attivo degli stagisti nello sviluppo dei progetti.

Oltre alle competenze tecniche, ho sviluppato anche abilità trasversali preziose. La comunicazione all'interno del team e la capacità di collaborare in un ambiente lavorativo sono state competenze che ho affinato grazie alle interazioni quotidiane con i colleghi. Inoltre, l'opportunità di lavorare a stretto contatto con tecnologie di punta,

come *Swift* e *SwiftUI*, ha ampliato il mio orizzonte professionale e mi ha preparato ad affrontare sfide più complesse nel futuro.

Nel complesso, il mio stage presso Rawfish è stato un periodo di crescita e apprendimento che mi ha fornito le basi solide per una carriera nel campo dello sviluppo di applicazioni e tecnologie innovative. Sono grato all'azienda per avermi offerto questa opportunità e sono entusiasta di portare con me le conoscenze, le competenze e le esperienze acquisite mentre mi muovo verso le prossime sfide professionali.

Acronimi e abbreviazioni

AES Advanced Encryption Standard. 63

API Application Program Interface. 63

CRUD Create Read Update Delete. 64

IDE Integrated Development Environment. 65

JSON JavaScript Object Notation. 65

SDK Software Development Kit. 66

UI/UX User Interface/User Experience. 67

Glossario

AES *AES* (*Advanced Encryption Standard*) è uno standard di crittografia a blocchi utilizzato per la cifratura di dati. 26, 61

API in informatica con il termine *Application Programming Interface, API* si indica ogni insieme di procedure, di solito raggruppate in set di strumenti specifici che permettono la comunicazione tra due software diversi. 7, 61

APIKEY Una *APIKEY* è una chiave che viene utilizzata per l'autenticazione di un utente all'interno di un sistema. 26

Apple TV *Apple TV* è un dispositivo prodotto da *Apple*, che permette di accedere a contenuti multimediali da Internet, come film, serie TV, musica, podcast, giochi, ecc..., e di riprodurli su un televisore di alta definizione. 1

Apple Watch *Apple Watch* è uno *smartwatch* prodotto da *Apple*. *Apple Watch* è un dispositivo indossabile che permette di accedere a contenuti multimediali da Internet, oltre a svolgere attività di *fitness tracking* e monitoraggio della salute. 1

App Store *App Store* è lo store di applicativi ufficiale distribuito da *Apple* per tutte le sue piattaforme. 7

AVPlayer *AVPlayer* è un *SDK* per la riproduzione di contenuti multimediali in *streaming* nativo per le piattaforme *Apple*. 2

Background In informatica con il termine *background* si indica che un processo viene eseguito in modo asincrono rispetto al processo principale, in modo da eseguire delle operazioni senza bloccare il processo principale. 9

Binding In *Swift*, un *Binding* è un meccanismo che permette di sincronizzare due oggetti, in modo che quando il valore di uno cambia, anche il valore dell'altro rispecchia il cambiamento effettuato. 11

Business logic Con il termine *business logic* si indica la logica di *business* di un'applicazione, ovvero la logica che permette di implementare le funzionalità dell'applicazione. 9

CI/CD In informatica con il termine *CI/CD* si indica un insieme di pratiche e metodologie per la distribuzione di software. *CI/CD* è l'acronimo di *Continuous Integration/Continuous Delivery*. 7

Codebase In informatica con il termine *codebase* si indica l'insieme di codice sorgente di un progetto software. 25

CRUD In informatica con il termine *CRUD* si indica un insieme di operazioni di base che possono essere effettuate su un database o su un servizio web. *CRUD* è l'acronimo di *Create, Read, Update, Delete*. 11, 61

Debugger In informatica con il termine *debugger* si indica un programma che permette di eseguire un programma in modalità di *debug*, ovvero eseguire un programma passo passo, permettendo di analizzare il contenuto delle variabili, il contenuto della memoria, ecc.... 7

Dependency injection In informatica con il termine *dependency injection* si indica un *design pattern* che permette di separare la creazione di un oggetto dalla sua utilizzazione. Tramite questo pattern un oggetto (o funzione) riceve un oggetto (o funzione) di cui ha bisogno, senza doverlo creare internamente. Questo permette di rendere il codice più modulare e testabile. 10

Enumerazione In informatica con il termine *enumerazione* si indica un tipo di dato astratto che permette di definire un insieme di costanti, automaticamente numerate in modo incrementale, in base all'ordine di definizione o secondo un valore specificato. 26

Feedback aptico Il *feedback aptico* è un tipo di *feedback* che viene fornito all'utente tramite il senso del tatto, generalmente utilizzando delle vibrazioni. 49

Fetch In informatica con il termine *fetch* si indica l'operazione di recupero di dati da un server. 28

Gamification Con il termine *gamification* si indica l'utilizzo di elementi tipici dei giochi/videogiochi in contesti diversi da essi, come ad esempio applicazioni per la salute, per incentivare l'utente a raggiungere determinati obiettivi. 40

Git *Git* è un *software* di controllo versione distribuito, utilizzabile da interfaccia a riga di comando, creato da Linus Torvalds nel 2005. 7

HealthKit *HealthKit* è un *framework* per la creazione di applicazioni per la gestione dei dati relativi alla salute dell'utente, presente su piattaforme *Apple*. *HealthKit* permette di accedere ai dati relativi alla salute dell'utente, come dati relativi a

fitness, attività fisica, nutrizione, sonno, ecc. . . e dati biometrici tramite l'utilizzo dei sensori di *Apple Watch* e *iPhone*. 1

IDE In informatica con il termine *Integrated Development Environment, IDE* (ing. ambiente di sviluppo integrato) si indica un *software* che permette di sviluppare applicazioni per una specifica piattaforma software, e/o hardware, e/o sistema operativo. Un IDE include un insieme di strumenti come compilatori, librerie, debugger, ecc. . . . 2, 61

iOS *iOS* è un sistema operativo sviluppato da *Apple* per *iPhone*. 1

iPadOS *iPadOS* è un sistema operativo sviluppato da *Apple* per *iPad*. È derivato da iOS. 6

Issue Con il termine *issue* si indica un problema, un bug, una richiesta di funzionalità, ecc. . . , che viene segnalato all'interno di un repository. 7

Issue tracker Un *issue tracker* è un *software* che permette di gestire le *issue* all'interno di un *repository*. 8

JSON JSON (JavaScript Object Notation) è un formato utilizzato per lo scambio di dati in applicazioni client-server. 24, 61

Keychain Nell'ambito delle tecnologie *Apple*, *Keychain* è un *container* criptato che permette di salvare in modo sicuro dati sensibili, come *password*, *token*, nomi utente, codici PIN, ecc. . . . 24

Mac *Mac* (abbreviazione di *Macintosh*) è una famiglia di *personal computer* prodotti da *Apple*. 7

macOS *macOS* (precedentemente *Mac OS X* e successivamente *OS X*) è un sistema operativo sviluppato da *Apple* per i computer *Macintosh*. È derivato da *NeXTSTEP*, un sistema operativo sviluppato da *NeXT*, che *Apple* acquisì nel 1996. 6

Memory leak In informatica con il termine *memory leak* si indica un errore di programmazione che si verifica quando un programma non riesce a liberare una porzione di memoria che non è più utilizzata. Questo errore può portare a un consumo eccessivo di memoria, e in certi casi a un crash del programma. 6

Mockup Il termine *mockup* indica un prototipo di un'interfaccia utente, creato generalmente da dei *designer* e utilizzato come punto di partenza per lo sviluppo in codice dell'interfaccia. 14

Polling Con il termine *polling* si indica una tecnica per la verifica di un evento, che consiste nel verificare periodicamente se l'evento è avvenuto. 26

Property wrapper In *Swift*, con il termine *property wrapper* si indica un tipo che funge da *wrapper* per un altro tipo in modo da aggiungere delle funzionalità al tipo originale. Un esempio è `@State` che permette di aggiungere la funzionalità di *reactivity* a una variabile, ovvero di aggiornare automaticamente la UI quando il valore della variabile cambia. 10

Push notification Una *push notification* è un messaggio che viene inviato da un server ad un client, che non ha richiesto esplicitamente il messaggio, in modo da notificare l'utente di un evento. 10

QRCode Un *QRCode* è un codice a barre bidimensionale che permette di codificare una serie di dati in una matrice di punti. 14

Rendering Il *rendering* è il processo di generazione di un'immagine digitale a partire da una descrizione matematica di una scena. 48

Repository In informatica con il termine *repository* si indica un archivio di dati, che contiene metadati e dati relativi a un progetto software. 7

SDK in informatica con il termine *Software Development Kit SDK* (ing. kit di sviluppo *software*) si indica un insieme di strumenti per lo sviluppo e la documentazione di *software* per una specifica piattaforma *software*, e/o *hardware*, e/o sistema operativo. Solitamente un *SDK* include un insieme di strumenti come compilatori, librerie, *debugger*, ecc. . . , e un insieme di documentazione come guide, esempi, ecc. . . . 2, 61

Seek Il termine *seek* indica l'operazione di spostamento all'interno di un contenuto multimediale, come un video o un brano musicale, in modo da riprodurre il contenuto a partire da un punto specifico. 20

Smart TV Una *Smart TV* è un televisore che, oltre alla visione dei normali canali televisivi, permette di accedere a contenuti multimediali da Internet, come film, serie TV, musica, podcast, giochi, ecc. . . . 15

Smartwatch Uno *smartwatch* è un dispositivo indossabile che permette di accedere a contenuti multimediali da Internet, oltre a svolgere attività di *fitness tracking* e monitoraggio della salute. 33

SQLite *SQLite* è una libreria open-source scritta in linguaggio C che implementa database engine SQL serverless, transazionale e che non richiede configurazione. 6

Streaming Con il termine *streaming* si indica una tecnica per la trasmissione di dati multimediali, come audio e video, che permette di riprodurli senza aver necessariamente già ricevuto l'intera trasmissione. 15

Struct In informatica con il termine *struct* si indica un tipo di dato astratto che permette di raggruppare più dati in un'unica struttura. 9

Swift *Swift* è un linguaggio di programmazione *object-oriented* e funzionale per sistemi *macOS*, *iOS*, *iPadOS*, *watchOS* e *tvOS*. È stato presentato da *Apple* durante la *WWDC* 2014. 1

Swift Package Manager *Swift Package Manager* è un gestore di dipendenze per il linguaggio di programmazione *Swift*. 25

SwiftUI *SwiftUI* è un *framework* per la creazione di interfacce utente per le piattaforme *Apple*. *SwiftUI* utilizza una sintassi dichiarativa, che permette di indicare cosa deve essere visualizzato, senza dover specificare come deve essere visualizzato. 1

Thumbnail Con il termine *thumbnail* si indica un'immagine di piccole dimensioni che viene utilizzata per identificare un contenuto multimediale, come un video o un brano musicale. 19

Token Con il termine *token* si indica una stringa di caratteri che viene utilizzata per l'autenticazione di un utente all'interno di un sistema. 9

Token JWT Un *token JWT* è un *token* che viene utilizzato per l'autenticazione di un utente all'interno di un sistema, e viene generato utilizzando lo standard *JSON Web Token*. 24

tvOS *tvOS* è un sistema operativo sviluppato da *Apple* per *Apple TV*. 1

UIKit *UIKit* è un *framework* per la creazione di interfacce utente per le piattaforme *Apple*. *UIKit* utilizza una sintassi imperativa, che descrive come i componenti visuali devono essere visualizzati. 6

UI/UX Con il termine *UI/UX* si indica l'insieme di pratiche e metodologie per la progettazione di interfacce utente. *UI/UX* è l'acronimo di *User Interface/User Experience*. 1, 61

UserDefaults In *Swift*, *UserDefaults* è un *container* che permette di salvare dati in modo persistente. 27

watchOS *watchOS* è un sistema operativo sviluppato da *Apple* per *Apple Watch*. È derivato da *iOS*, ma a differenza di questo, *watchOS* è un sistema operativo per dispositivi indossabili. 6

Xcode *Xcode* è un *IDE* di sviluppo per applicazioni *iOS*, *iPadOS*, *macOS*, *watchOS* e *tvOS*. *Xcode* è stato sviluppato da *Apple*. 2

Bibliografia

Siti web consultati

[1] *Build device-to-device interactions with Network Framework - WWDC22 - Videos - Apple Developer*. URL: https://developer.apple.com/videos/play/wwdc2022/110339/ (cit. a p. 42).

[2] *Combine | Apple Developer Documentation*. URL: https://developer.apple.com/documentation/combine (cit. a p. 6).

[3] *Core Data | Apple Developer Documentation*. URL: https://developer.apple.com/documentation/coredata/ (cit. a p. 6).

[4] *Designing for tvOS - Apple*. URL: https://developer.apple.com/design/human-interface-guidelines/designing-for-tvos (cit. a p. 14).

[5] *Designing for watchOS - Apple*. URL: https://developer.apple.com/design/human-interface-guidelines/designing-for-watchos (cit. a p. 35).

[6] *Fork - a fast and friendly git client for Mac and Windows*. URL: https://git-fork.com/ (cit. a p. 8).

[7] *Git*. URL: https://git-scm.com/ (cit. a p. 7).

[8] *Linear - A better way to build products*. URL: https://linear.app/ (cit. a p. 8).

[9] *Platform | GitLab*. URL: https://about.gitlab.com/platform/?stage=plan (cit. a p. 7).

[10] *Roku vs. Chromecast vs. Apple TV vs. Fire TV: What's Best for Netflix?* URL: https://variety.com/2015/digital/news/roku-chromecast-apple-tv-fire-tv-best-netflix-streamer-1201650291/ (cit. a p. 15).

[11] *Swift - Apple Developer*. URL: https://developer.apple.com/swift/ (cit. a p. 6).

[12] *SwiftUI | Apple Developer Documentation*. URL: https://developer.apple.com/documentation/swiftui/ (cit. a p. 6).

[13] *What is Postman? Postman API Platform.* URL: https://www.postman.com/product/what-is-postman/ (cit. a p. 7).

[14] *Workouts - Apple Developer Documentation.* URL: https://developer.apple.com/design/human-interface-guidelines/workouts (cit. a p. 35).

[15] *Xcode | Apple Developer Documentation.* URL: https://developer.apple.com/documentation/xcode (cit. a p. 7).