

Programación Multimedia y Dispositivos Móviles

UP4.2 - Infinite Scroll y Detalles

Autor: Antonio Calabuig Puigvert y Sebastián Villa Ponce

Centro: IES Salvador Gadea

Departamento: Informática

Ciclo: Desarrollo de Aplicaciones Multiplataforma

Fecha: 2025-09-14



UP4.2 - Infinite Scroll y Detalles

- Programación Multimedia y Dispositivos Móviles
 - UP4.2 – Infinite Scroll y Detalles
- UP4.2 – Infinite Scroll y Detalles
- 1. Introducción
- 2. Determinar el fin del scroll en un FlatList
- 3. TanStack – InfiniteQuery
- 4. Pantalla - Detalles de la película
- 5. useMovie - TanStack
- 6. Mostrar detalles de la película
- 7. Gradiente sobre el poster
- 8. Descripción de la película
- 9. Tarea: Implementación del reparto de actores de la película
 - Parte 1 - Lógica de negocio
 - Parte 2 - UI
 - Importante
- Bibliografía

1. Introducción

Continuando con la aplicación desarrollada en la sección anterior, en esta sección nos centraremos en el desarrollo de la pantalla por id. Es decir, de los detalles de las películas. Seguiremos trabajando también elementos cargados en caché y dejando la aplicación lo más pulida posible. Cabe destacar los elementos principales que veremos:

1. Caché con TanStack
2. Actores
3. Detalle
4. Stack
5. Custom Hooks
6. Gradients
7. Interfaces y nuevas acciones

Reconstruimos la aplicación y empezamos.

2. Determinar el fin del scroll en un FlatList

La intención que tenemos en este desarrollo es detectar cuando llegamos al final de la lista para, a continuación cargar las siguientes películas. Tiene que dar la sensación de scroll infinito hasta que ya no quede nada por cargar.

En el fichero [presentation/components/movies/MovieHorizontalList](#), en las propiedades del FlatList existe un método que es llamado cada vez que el scroll se activa y se llama [onScroll](#).

Este elemento es muy ruidoso ya que emite muchos valores, un scroll dispara muchas acciones y vamos a ver qué muestra para hacernos una idea.

```
<FlatList
    horizontal
    showsHorizontalScrollIndicator={false}
    nestedScrollEnabled={true}
    data={movies}
    keyExtractor={({ item }) => `${item.id}`}
    renderItem={({ item }) =>
        <MoviePoster id={item.id} poster={item.poster} smallPoster />
    }
    onScroll={({ event }) => {
        console.log(event);
    }} />
```

Comprobad el scroll en el emulador y veréis la cantidad de información que emite. Motivo por el que tenemos que filtrar el contenido de toda esa información. Muestra posición actual, donde me encuentro en el scroll. Afecta tanto a un scroll horizontal como vertical.

Miramos el tipo de dato del evento: `event`:

`NativeSyntheticEvent<NativeScrollEvent>`, copiamos este tipo de dato para crear una función y no tener lógica esparcida por dentro del componente.

```
const onScroll = (event: NativeSyntheticEvent<NativeScrollEvent>) =>
{};


```

Realicemos los imports correspondientes. ¡Muy importante!

Como sabemos que este evento emite mucha información tenemos que evitar que se dispare si en algún momento se está cargando información, por lo que vamos a crear una variable local `isLoading` para parar la emisión de información del evento.

```
const isLoading = useRef(false);

const onScroll = (event: NativeSyntheticEvent<NativeScrollEvent>) =>
{
    if (isLoading.current) return;
};

//Mandamos a llamar la función en la propiedad onScroll del FlatList
...
<FlatList
    ...
    onScroll={onScroll} />


```

Para determinar el final del scroll, necesitaremos desestructurar algunas variables que viene dentro del evento. Veamos como resultaría y como las usariamos.

Al mismo tiempo comprobaremos si hemos llegado al final y emitiremos un evento de consola para indicar que hemos llegado. Justo en este momento sería donde volveríamos a realizar la llamada de carga a la siguiente tanda de elementos.

```

const onScroll = (event: NativeSyntheticEvent<NativeScrollEvent>) =>
{
  if (isLoading.current) return;

  const {
    // posición actual en el scroll
    contentOffset,
    // lo que se ve en pantalla
    layoutMeasurement,
    // contenido de lo que tenemos
    contentSize } = event.nativeEvent;

  // calcular si estamos cerca del final del scroll
  const isEndReached = contentOffset.x + layoutMeasurement.width +
600 >= contentSize.width;

  // Nada que hacer si no hemos llegado al final
  if (!isEndReached) return;

  // Aquí cargaríamos más películas y paramos que se vuelva a
  disparar
  isLoading.current = true;

  // Simular carga
  console.log('Cargar más películas...');

};

```

¿Como llevaríamos a cabo ahora la nueva llamada? Estaría bien tener una tarea asíncrona que pudieramos llamar para cargar la siguiente parte, etc. Pero para ello tenemos **TanStack** que nos ayudará a gestionar toda la problemática del InfiniteScroll.

Para ello, en el interface añadiremos una nueva función llamada loadNextPage, que será opcional. En caso de venir informada en la instancia del componente, la mandaremos llamar.

```

interface Props {
  title?: string;
  movies: Movie[];
  className?: string;
  loadNextPage?: () => void;
}

```

En la función onScroll, a continuación del console.log(...), comprobamos si viene informado y llamamos.

```
...
    // Simular carga
    console.log('Cargar más películas...');
    loadNextPage && loadNextPage();
...

```

3. TanStack - InfiniteQuery

Veamos ahora cuando se da la situación de haber llegado al final del scroll, cómo mandar a llamar a los nuevos datos. Para no repetir la implementación por cada una de las acciones, vamos a centrarnos únicamente en una. Pongamos, por ejemplo la de `top-rated.action.ts`. **Para el resto será lo mismo por lo que vosotros ahora en clase podeís implementar el resto.**

Para ello, lo primero será crear un nuevo objeto en el que recibiremos las opciones de la llamada, como la página y el límite de elementos en la llamada. Éstos no nos los inventamos, dentro del ejemplo de la llamada a la api podemos ver que en los query params, existen estos dos parámetros. [Top Rated](#)

```
interface Options {
  page?: number;
  limit?: number;
}
```

Pasamos este objeto desestructurado a la interfaz del action que tenemos ya implementado.

```
export const topRatedMoviesAction = async ({ page = 1, limit = 10 }: Options) => {
  ...
};
```

Como sabemos, en `axios`, cuando realizamos una llamada, tiene un parámetros `params` en el que podemos especificar query params que podmeos enviar en la llamada a un endpoint de una api. *Esto ya lo vimos en la UP1 en la Introducción a Typescript cuando montamos la tabla de usuarios.*

```
const { data } = await movieApi.get<MovieDBMoviesResponse>('/top_rated', {
    params: {
        page,
    }
});
```

Ahora en el **hook** `useMovies.ts` tendremos un error en la función de la llamada al endpoint de `top_rated`. Esta parte vamos a ver dos opciones para resolverlo. Una estática y otra dinámica que no nos permitirá el uso de `useQuery(...)`.

Versión estática:

```
const topRatedQuery = useQuery({
    queryKey: ['movies', 'topRated', 1],
    queryFn: () => topRatedMoviesAction({ page: 1 }),
    staleTime: 1000 * 60 * 60 * 24, // 24 hours
});
```

En este caso, pasamos el parámetro `page` ad-hoc y también mapeamos la key para saber lo que corresponde con la cache de 'movies', 'topRated' página 1.

Para el caso que nos atañe, resulta que **TanStack** dispone de una función a parte de `useQuery` que se llama `useInfiniteQuery`. El uso es el mismo pero añade más propiedades a la configuración de la llamada.

Uno de las propiedades es el `initialPageParam` que sería nuestro valor por defecto en la carga de datos. A parte, en la llamada a la función que lanza la llamada al endpoint, el `queryFn`, desestructuraremos el parámetro `pageParam` para poder pasarlo nosotros de forma dinámica.

Otra propiedad de `useInfiniteQuery` es el `getNextPageParam` que es una función que tiene como parámetros la última pagina mostrada `lastPage` y el listado de películas `pages` que es del tipo `Movie[][]`, lo que significa que es una array de arrays con el siguiente formato `[[movie, movie, movie], [movie, movie, movie]]` donde `movie` es un objeto del tipo `Movie`.

```
const topRatedQuery = useInfiniteQuery({
  initialPageParam: 1,
  queryKey: ['movies', 'topRated'],
  queryFn: ({ pageParam }) => {
    return topRatedMoviesAction({ page: pageParam })
  },
  staleTime: 1000 * 60 * 60 * 24, // 24 hours
  getNextPageParam: (lastPage, pages) => pages.length + 1,
});
```

Este funcionamiento a partir de aquí es un poco diferente a como tenemos la implementación de carga de los `MovieHorizontalList`. Vamos a `app/home/index.tsx` y veremos que tenemos un error en la carga de la lista de las pelis Top Rated, solo tenemos que modificar el acceso a los datos porque recordad que lo que devuelve la `infiniteQuery` es una array de arrays.

```
{/* Top Rated */}
<MovieHorizontalList title='Mejor Valoradas' movies={topRatedQuery.data?.pages.flat() ?? []}
  className='mb-5' />
```

`flat()` Aplana un *array de arrays* que significa unificar los arrays anidados en uno solo y de nivel superior, eliminando los niveles de anidamiento para obtener una lista simple.

Probamos en el simulador que todo funciona de forma correcta. Revisamos el log en la consola en el que indicara el `pageParam` y el mensaje de `Cargado siguientes películas`.

Es el momento ahora de aplicar la llamada sucesiva para que siga cargando más información. ¿Recordáis que en la definición de la interfaz del componente `MovieHorizontalList` habíamos creado una función llamada `loadNextPage`? Pues bien, será en este parámetro del componente donde le pasaremos la función que tiene que ejecutar para seguir cargando más páginas. Esta función pertenece a `useInfiniteQuery` y se llama `fetchNextPage`.

```
/* Top Rated */
<MovieHorizontalList
  title='Mejor Valoradas'
  movies={topRatedQuery.data?.pages.flat() ?? []}
  className='mb-5'
  loadNextPage={topRatedQuery.fetchNextPage} />
```

Cerramos toda aplicación en ejecución en nuestro simulador, la lanzamos de nuevo npm run start -c para limpiar cachés y lanzamos la aplicación en el simulador a ver qué pasa.

Al llevarlo al simulador, carga la primera página, carga la segunda, pero después ya no continua la carga. Esto es un problema de la implementación de la función `onScroll` que tenemos implementada en el componente `MovieHorizontalList`. La variable `isLoading`, después de la primera carga (la de la página 2), no ha vuelto a `false` para poder permitir una siguiente carga.

Tenemos que manejar en qué puntos después de realizar una carga, ésta (la variable `isLoading`), vuelve a ser true/false dependiendo de la acción que estemos ejecutando.

Lo correcto sería marcarla a false después de ejecutar la línea de:

```
// Simular carga
console.log('Cargar más películas... ');
loadNextPage && loadNextPage();
```

¿Cómo lo vamos a hacer?

Una opción sería cambiar el valor de `isLoading` a `false`.

```
// Simular carga
console.log('Cargar más películas... ');
loadNextPage && loadNextPage();

isLoading.current = false;
```

Alternativamente, podríamos usar un `useEffect` asociado al cambio del valor de la variable `movies`, para que pasado un tiempo vuelve a `false`.

```
useEffect(() => {
  setTimeout(() => {
    isLoading.current = false;
  }, 500);
}, [movies])
```

Para finalizar, resulta que si seguimos haciendo scroll sin parar mientras va cargando películas, veremos que da un error como este:

Encountered two children with the same key, `%. Keys should be unique so that components maintain their identity across updates. Non-unique keys may cause children to be duplicated and/or omitted – the behavior is unsupported and could change in a future version.

Resulta que es un fallo de implementación que nos llegan dos elementos con la misma key y la app crashea.

Para resolverlo, modificaremos el `keyExtractor` del `FlatList` en el componente de `MovieHorizontalList`.

```
keyExtractor={(item, i) => `${item.id}-${i}`}
```

Ñapa as a code

A veces no dependen de nosotros las implementaciones externas con las que trabajamos por lo que de vez en cuando es necesario adaptarse al medio.

4. Pantalla - Detalles de la película

Vamos a crear la pantalla de visualización del elemento movie. Para ello creamos un nuevo fichero en [app/movie/\[id\].tsx](#).

```
import React from 'react'
import { Text, View } from 'react-native'

const MovieScreen = () => {
  return (
    <View>
      <Text>MovieScreen</Text>
    </View>
  )
}

export default MovieScreen
```

Como la pantalla está dentro de un stack, nosotros podremos navegar a ella. Solo falta indicar desde donde lanzamos la navegación hacia la página de descripción de movie.

Entonces en el componente MoviePoster

([presentation/components/movies/MoviePoster.tsx](#)) el elemento principal del componente es un **Pressable**. A este Pressable, hay que añadirle la redirección.

```
...
<Pressable
  className={`active:opacity-90 px-2 ${className}`}
  onPress={() => router.push(`/movie/${id}`)}>
...

```

Esta redirección se hace con **push** y no con **replace** porque lo que queremos es poder volver atrás. Si reemplazamos con **replace** machamos la pila y no existirá un momento anterior.

Probamos en el simulador. En caso de no funcionar o no encontrar la ruta, recargamos la app desde la terminal y probamos de nuevo.

Ahora ya vamos a con la implementación de la pantalla [app/movie/\[id\].tsx](#). Para dicha implementación empezamos capturando el id de la pelicula.

```
const { id } = useLocalSearchParams();
```

En primer lugar observaremos tanto el endpoint de `/now_playing`.

Y ahora el de una movie concreta que sería el de detalles `/movie/{movie_id}`.

Con la respuesta que emite, la llamada a `/movie/{movie_id}` vamos a copiar el resultado y crearemos una interfaz en `infrastructure/interfaces/moviedb-movie.response.ts`. Con la ayuda de la extensión de VSCode "Pase JSON as Code", la invocamos y como RootName para la interfaz la llamaremos `MovieDBMovieResponse` y automáticamente se creará la interfaz asociada a la estructura de datos que devuelve la llamada al endpoint `/movie/{movie_id}`.

Como ya sabemos el endpoint al que tenemos que llamar, vamos ahora a implementar la `action` en la parte del `core` de nuestra aplicación.

Crearemos el fichero `core/actions/movie/get-movie-by-id.action` que como en los casos anteriores, tiene la llamada al endpoint de los detalles de la película que hemos visitado anteriormente.

```
export const getMovieByIdAction = async (id: number | string) => {
  try {
    const { data } = await movieApi.get<MovieDBMovieResponse>(`/movie/${id}`);
    console.log(data)

    return data;
  } catch (error) {
    console.log(error);
    throw "Cannot fetch now playing movies";
  }
}
```

Realizamos los imports necesarios.

Inicialmente, todavía no enviamos la información a la vista de forma enriquecida, veamos primero que al realizar la llamada, saca la información en crudo sobre la pantalla de descripción de película.

En `app/movie/[id].tsx` llamamos la función `getMovieByIdAction()` pasando como `id` el que hemos capturado al iniciar esta pantalla de forma temporal, esto no irá aquí a futuro.

```
getMovieByIdAction(+id)
```

El `+` se pone porque detecta que es un `string[]` cuando sabemos que el parámetro únicamente será un valor `string`. De esta forma, con `+` se le indica que lo que se pasa es un `string` y no un `string[]`.

Esto solo lo usaremos en determinados casos que no sea capaz de entender el parámetro que se pasa a la función cuando el parámetro es calculado a partir de `useLocalSearchParams()`.

Probamos en el simulador y comprobamos que nos devuelve la información a través de la consola.

Es momento ahora de realizar la implementación de un mapper y tratamiento de datos para seguir con la filosofía de nuestra implementación. Para ello lo primero será trabajar con una ampliación del objeto Movie. Entonces en la interfaz Movie que tenemos actualmente, vamos a crear otra nueva tal y como sigue:

```
export interface CompleteMovie extends Movie {
  genres: string[];
  duration: number;
  budget: number;
  originalTitle: string;
  productionCompanies: string[];
}
```

Añadimos nuevos campos a la ya existente información previa.

Volvemos a la implementación de la acción `core/actions/movie/get-movie-by-id.action` y lo primero que vamos a indicar es una Promise en la que forzamos el tipo de objeto que va a devolver al realizar la llamada al endpoint.

```
export const getMovieByIdAction = async (id: number | string): Promise<CompleteMovie> => {
    ...
}
```

Actualmente con lo que tenemos no podemos devolver este objeto porque nuestro **mapper** no está preparado. Para ello será necesario en el mapper de movie **infrastructure/mappers/movie.mapper.ts** crearemos otra función estática que nos devuelve el CompleteMovie con los nuevos campos.

```
static fromTheMovieDBToCompleteMovie = (movie: MovieDBMovieResponse): CompleteMovie => {
    return {
        id: movie.id,
        title: movie.title,
        description: movie.overview,
        rating: movie.vote_average,
        realeaseDate: new Date(movie.release_date),
        poster:
`https://image.tmdb.org/t/p/w500${movie.poster_path}`,
        backdrop:
`https://image.tmdb.org/t/p/w500${movie.backdrop_path}`,
        budget: movie.budget,
        duration: movie.runtime,
        // movie.genres es del tipo Genre[] y tiene dos atributos
        (id, name)
            // con un map sobre el campo movie.genres, transformamos el
            // objeto Genres[]
            // en un array de string de los nombres de los Genres
            // (géneros)
            genres: movie.genres.map(g => g.name),
            originalTitle: movie.original_title,
            // mismo caso que en el Genre pero en tipo de dato
            ProductionCompany[]
                productionCompanies: movie.production_companies.map(c =>
c.name),
            }
    }
```

Queda claro que esta función anteriormente implementada es optimizable ya que lo primero que podríamos hacer es que la función llame a la anterior para que devuelva una movie de tipo Movie y a continuación asociar los nuevos campos.

De esta forma queda más claro qué es lo que se hace y ya queda a elección del programador su mejora y optimización de código.

Ahora ya podemos volver de nuevo a `core/actions/movie/get-movie-by-id.action` para usar nuestro mapper:

```
export const getMovieByIdAction = async (id: number | string): Promise<CompleteMovie> => {
  try {
    const { data } = await movieApi.get<MovieDBMovieResponse>(`/${id}`);

    return MovieMapper.fromTheMovieDBToCompleteMovie(data);
  } catch (error) {
    console.log(error);
    throw "Cannot fetch now playing movies";
  }
}
```

Con esto ya tenemos nuestro objeto de datos listo y mapeado para poder empezar a diseñar la vista de Movie y optimizar su funcionamiento.

5. useMovie - TanStack

Vamos a optimizar la llamada usando TanStack como hemos hecho con las llamadas anteriores. Pero es importante saber con la implementación actual, cada vez que volvemos a la pantalla del listado de películas y accedo de nuevo a un recurso que previamente ya había accedido, la llamada se realiza de nuevo.

Esto en APIs que cobran por acceso, lo que estamos haciendo es consumir cuota de usuario y generalmente debemos optimizar esto también. De ahí que las caches tengan una duración de X horas dependiendo del tipo de datos que se trabajan.

Cuando los datos tienen una tendencia a ser estáticos, la duración debe ser mayor, en el caso de datos de tendencia dinámica o que pueden variar en función del tiempo, no usaremos cachés largas.

Un ejemplo de esto podría ser el siguiente:

1. **Un post de twitter, puede cambiar poco o nada en el tiempo** --> Caché de larga duración.
2. **Los likes, retweets, reposts, replies (valores y contenidos) de un post cambian de forma dinámica a medida que están más tiempo publicados.** --> Caché de corta duración o directamente sin caché.

En un ejemplo como el anterior, habrían dos llamadas diferentes. La que trae el post y por otra parte la que trae los elementos variables de un post.

Al igual que hicimos un hook que se encargaba de la optimización de los listados de las movies, vamos a hacer lo mismo para una movie. Por tanto, creamos el hook `useMovie()`. Con el atajo `rafc`, creamos un *react functional component*.

```
export const useMovie = (id: number | string) => {
  const movieQuery = useQuery({
    queryKey: ['movie', id],
    queryFn: () => getMovieByIdAction(id),
    staleTime: 1000 * 60 * 60 * 24,
  })

  return {
    movieQuery
  }
}
```

Volvemos de nuevo a la pantalla de personalización de movie `app/movie/[id].tsx` y actualizamos la llamada. Borramos la llamada a la action y hacemos llamar al hook.

```
const { movieQuery } = useMovie(+id);
```

Puede que con una conexión de alta latencia, la información tarde en cargar. Pues bueno, los resultados de un TanStack, tiene un set de atributos con los que se puede trabajar. En este caso, podemos comprobar el estado de la llamada y ver si aún está cargando. Si es el caso, mostraremos hasta que acabe de cargar un indicador de carga.

```
if (movieQuery.isLoading) {
    return (
        <View className='flex flex-1 justify-center items-center'>
            <Text className='mb-4'>Espere por favor</Text>
            <ActivityIndicator color='purple' size={30} />
        </View>
    )
}

return (
    <ScrollView>
        <Text>{movieQuery.data?.title ?? 'No tiene'}</Text>
    </ScrollView>
)
```

Podemos evitar el uso de `data?` a la hora de leer los parámetros si en el `if` del loading, comprobamos también que la data no esté vacía. `if (movieQuery.isLoading || !movieQuery.data)`

Podemos comprobar en el simulador que funciona. Si la conexión es rápida no se verá, podemos comentar unos instantes la comprobación para ver que lo muestra y posteriormente la devolvemos a su estado natural.

6. Mostrar detalles de la película

Bien, es momento ya de maquetar la vista descriptiva de una película a partir de los datos que tenemos. Este desarrollo lo dividiremos en tres partes, o lo que es lo mismo en tres componentes diferentes, ya que uno de ellos es el reparto de la película que es otra petición diferente que realizaremos ya en la propia vista de descripción de película.

Creamos la carpeta movie dentro de `presentation/components/movie` y dentro el componente `MovieHeader.tsx`.

El return del MovieHeader, lo inicializamos en modo contenedor porque va a contener varios componentes. `<>...</>`. Uno será la image, otro el display de los textos de titulos y un botón que permitirá la vuelta atrás.

```

interface Props {
  poster: string;
  originalTitle: string;
  title: string;
}

const MovieHeader = ({ poster, originalTitle, title }: Props) => {

  const { height: screenHeight } = useWindowDimensions();

  return (
    <>
      <View style={{{
        position: 'absolute',
        zIndex: 99,
        elevation: 9,
        top: 40,
        left: 10,
      }}}>
        <Pressable onPress={() => router.dismiss()}>
          <Ionicons name='arrow-back' size={30} color='white' className='shadow' />
        </Pressable>
      </View>
      <View style={{ height: screenHeight * 0.7 }} className='shadow-xl shadow-black/20'>
        <View className='flex-1 rounded-b-[25px] overflow-hidden'>
          <Image source={{ uri: poster }} resizeMode='cover' className='flex-1' />
        </View>
        <View className='px-5 mt-5'>
          <Text className='font-normal'>{originalTitle}</Text>
          <Text className='font-semibold text-2xl'>{title}</Text>
        </View>
      </View>
    )
}

export default MovieHeader

```

Esto dará como resultado algo así:



Zootopia 2

Zootrópolis 2

Podemos probar el resultado en el simulador. Y testear el botón de vuelta atrás. Todo es correcto, pero inicialmente un problema que ahora no se ve y que puede afectar es que cuando el fondo del poster sea blanco, no se divisará de forma correcta el botón de vuelta atrás. Con lo que vamos a trabajar también con un gradiente en la siguiente sección para poder visibilizar sobre el top el botón de vuelta atrás.

7. Gradiente sobre el poster

Para poder realizar el LinearGradient, os dejo el [enlace](LinearGradient. (s. f.). Expo Documentation. <https://docs.expo.dev/versions/latest/sdk/linear-gradient/>) a la documentación oficial de Expo.

Para poder usarlo debemos instalar esta dependencia en el proyecto:

```
npx expo install expo-linear-gradient
```

Según la documentación esta sería una forma correcta de aplicación:

```
import { LinearGradient } from 'expo-linear-gradient';

<LinearGradient
  // Background Linear Gradient
  colors={['rgba(0,0,0,0.8)', 'transparent']}
  style={styles.background}
/>
```

Ahora lo aplicamos sobre nuestro componente. Lo principal realizar el import de LinearGradient, que lo tenemos en el ejemplo anterior y también en la documentación oficial.

```
<LinearGradient colors={['purple', 'green']} style={{
  height: screenHeight * 0.4,
  position: 'absolute',
  zIndex: 1,
  width: '100%',
}} />
```

Lo lanzamos en el emulador y vemos con este ejemplo hace un degradado de morado a verde. Realmente lo que nosotros es un degradado con transparencia para que no oculte la imagen y se vea más oscuro arriba.

Nuestra intención principal es visibilizar el botón, con lo que el gradiente podríamos empezarlo desde la esquina superior izquierda hasta la esquina inferior derecha. Con la propiedad `start={[0, 0]}` del LinearGradient, cambia el sentido del gradiente.

Ahora aplicamos el cambio de color y la posición y ya quedará perfecto.

```
<LinearGradient
    colors={['rgba(0,0,0,0.3)', 'transparent']}
    start={[0, 0]}
    style={{
        height: screenHeight * 0.4,
        position: 'absolute',
        zIndex: 1,
        width: '100%',
    }} />
```

Se podría visualizar algo tal que así:



8. Descripción de la película

Para esta parte, crearemos un nuevo componente. Podríamos seguir implementando todo en el anterior componente, pero de esta forma quedaría más estructurado y mantenible. Esto también es decisión del desarrollador de cuanto quiera modular su aplicación o agrupar contenidos.

El componente, irá dentro de la carpeta `presentation/components/movie` y lo llamaremos `MovieDescription.tsx`. Lo inicializamos.

En este componente como interface Props le pasaremos el objeto movie completo. De forma que dándole estilos quedará algo tal que así:

```
import { Formatter } from '@helpers/formatter';
import { CompleteMovie } from '@infrastructure/interfaces/movie.interface';
import React from 'react';
import { Text, View } from 'react-native';

interface Props {
    movie: CompleteMovie;
}

const MovieDescription = ({ movie }: Props) => {
    return (
        <View className='mx-5'>
            <View className='flex flex-row'>
                <Text>{movie.rating}</Text>
                <Text> - {movie.genres.join(',')}</Text>
            </View>

            <Text className='font-bold mt-5 text-2xl'>Sinopsis</Text>
            <Text className='font-normal mt-2'>{movie.description}</Text>

            <Text className='font-bold mt-5 text-2xl'>Presupuesto</Text>
            <Text className='font-normal mt-2 mb-10'>
                {Formatter.currency(movie.budget)}</Text>
            </View>
        )
}

export default MovieDescription
```

Cabe destacar que un elemento en toda esta implementación y es el apartado del budget. Este apartado, resulta que es una cantidad de dinero y que viene indicada como un número entero de dinero. Para poder darle un formato en moneda, existe una forma ya implementada sobre TypeScript que es la clase Intl sobre internacionalización.

Para resolución de este elemento hemos creado una carpeta **helpers** en el root del proyecto donde implementamos un class con un método para formatear cantidades de dinero.

En la carpeta **helpers** creamos el fichero **formatter.ts** en el que realizaremos la implementación del formateador de moneda:

```
export class Formatter {

    public static currency(value: number): string {
        return new Intl.NumberFormat('en-US', {
            style: 'currency',
            currency: 'USD',
        }).format(value)
    }
}
```

A la hora de mostrar la cantidad del buget de la película importamos el Formatter y lo llamamos para formatear el número.

```
Formatter.currency(movie.budget)
```

9. Tarea: Implementación del reparto de actores de la película

Con los conocimientos adquiridos a lo largo de esta UP, debéis realizar un componente llamado MovieCast que monte un FlatList horizontal después de los dato de la película en el que al igual que se muestran las películas en la home, muestre aquí el reparto de actores de la película.

Parte 1 - Lógica de negocio

1. Crear interfaz Cast

```
export interface Cast {  
  id: number;  
  name: string;  
  character: string;  
  avatar: string;  
}
```

2. Crear interfaz CreditsResponse, para identificar la respuesta de [MovieCredits](#)
3. Crear acción para obtener los actores [getMovieCastAction](#), debe de recibir [movieId: number](#), y retornar una [Promesa](#) que resuelve un array de [Cast](#).
4. Crear un mapper para transformar la respuesta de la API a un array de Cast

Ejemplo:

```

import { Cast } from '../interfaces/movie/cast.interface';
import { MovieDBCast } from '../interfaces/the-movie-
db/credits.response';

export class CastMapper {
  static fromMovieDBCastToEntity(actor: MovieDBCast): Cast {
    return {
      id: actor.id,
      name: actor.name,
      character: actor.character ?? 'No character',
      avatar: actor.profile_path
        ? `https://image.tmdb.org/t/p/w500${actor.profile_path}`
        : 'https://i.stack.imgur.com/l60Hf.png', // esto en caso
      de no tener imagen
    };
  }
}

```

5. En el custom hook `useMovie`, conectar TanStack query para obtener los actores de la película y retornar el query.

Parte 2 - UI

1. Crear componente `MovieCast`, que recibe un array de `Cast` y renderiza la lista de actores dentro de un `FlatList`.
2. Crear componente `ActorCard`, que recibe un `Cast` y renderiza la información del actor, pueden usar el siguiente código como base:

```
import { Image, Text, View } from 'react-native';
import { Cast } from
'@/infrastructure/interfaces/movie/cast.interface';

interface Props {
    actor: Cast;
}

export const ActorCard = ({ actor }: Props) => {
    return (
        <View className="mx-10 w-[60px]">
            <Image
                source={{ uri: actor.avatar }}
                className="w-[100px] h-[150] rounded-2xl shadow"
                resizeMode="cover" />

            <View>
                <Text
                    numberOfLines={2}
                    adjustsFontSizeToFit
                    className="font-bold text-lg">
                    {actor.name}
                </Text>
                <Text className="text-gray-600 text-xs">
                    {actor.character}</Text>
                </View>
            </View>
        );
};
```

Importante

Si se presenta un error del GestureHandler, debéis colocar el GestureHandlerRootView en el _layout principal de la aplicación en el directorio /app

```
import { View, Text } from 'react-native';

import { QueryClient, QueryClientProvider } from '@tanstack/react-
query';

import '../global.css';
import { Stack } from 'expo-router';
import { GestureHandlerRootView } from 'react-native-gesture-
handler';

const queryClient = new QueryClient();

const RootLayout = () => {
  return (
    <GestureHandlerRootView>
      <QueryClientProvider client={queryClient}>
        <Stack
          screenOptions={{
            headerShown: false,
          }}
        />
        </QueryClientProvider>
      </GestureHandlerRootView>
    );
};

export default RootLayout;
```

Bibliografía

Getting started. (s. f.). The Movie Database (TMDB).

<https://developer.themoviedb.org/docs/getting-started>

Movie Top Rated List. (s. f.). The Movie Database (TMDB).

<https://developer.themoviedb.org/reference/movie-top-rated-list>

/now_playing <https://developer.themoviedb.org/reference/movie-now-playing-list>

/movie/{movie_id} <https://developer.themoviedb.org/reference/movie-details>

LinearGradient. (s. f.). Expo Documentation. <https://docs.expo.dev/versions/latest/sdk/lineargradient/>

/movie/{movie_id}/credits <https://developer.themoviedb.org/reference/movie-credits>