

Bargain Games

Autor: Víctor Béjar Martín

Tutor: Javier Martín Rivero

Fecha de entrega: 28/05/2025

Convocatoria: 2024 2025

Introducción	2
Motivación	2
Abstract	2
Metodología utilizada	3
Tecnologías y herramientas	3
Estimación de recursos y planificación:	4
Análisis	5
Diseño	5
Diagramas	6
Documentación	11
Pruebas	27
Conclusiones	27
Vías futuras	28
Bibliografía/Webgrafía	28

Introducción

Un proyecto de Android desarrollado con la Api de steam para buscar tus juegos favoritos y deseados, ver sus detalles y recibir notificaciones de sus ofertas y noticias. La aplicación cuenta con un buscador y unos juegos recomendados cuando se seleccionen podrás ir a una vista más detallada del juego. Ahí puedes añadirlo tanto a favoritos como a deseados, luego si cambias de página a la de guardados puedes revisar qué juegos tienes en cada uno y te llegarán notificaciones de novedades y ofertas de estos juegos.

Motivación

Steam muchas veces no te notifica correctamente los juegos en oferta de tu lista de deseados. Además la visualización de noticias no es siempre la más correcta y no te avisa tampoco de ellas. Por eso esta app servirá para que te avise correctamente de estás novedades.

También como al tener tu sesión iniciada en steam busca los juegos según tus preferencias, suele ocultar el juego que en realidad estás buscando en ese instante, siendo molesto cuando buscas información específica.

Abstract

Steam is the leading platform for video games among me and my friends. However, despite its popularity, it has a major weakness: the poor communication of news and discounts related to the games users care about. Because of this, I decided to develop an Android app that improves this experience by keeping players informed about their favorite titles.

The app will allow users to add games to a favorites list and receive notifications when there are news updates. Additionally, if a game is added to the wishlist, users will be alerted when the game goes on sale.

Beyond notifications, the app will offer features such as searching for games, viewing detailed descriptions, trailers, images, and the latest news articles directly inside the app.

My main motivation is to create a simple and efficient tool that makes it easier for gamers to stay updated and take advantage of the best offers without having to manually search for them. I believe that by improving the way news and discounts are

delivered, users can have a better experience and enjoy their favorite games even more.

Objetivos propuestos

Generales:

- Una búsqueda de juegos de steam.
- Poder ver la lista de juegos deseados / favoritos.
- Notificación de noticias y ofertas.

Específicos:

- Poder marcar juegos como favoritos.
- Poder marcar juegos como deseados.
- Visualización de la información de juegos correcta.
- Ver ofertas de juegos.
- Ver información de noticias.

Metodología utilizada

KANBAN

Backlog:

- Obtener datos del juego por la api de steam.
- Convertir datos en card.
- Poner filtros sencillos de usar en el buscador del usuario.
- Notificaciones al usuario.
- Saber cuando un juego está en oferta para mandar notificación.
- Saber cuando un juego tiene una noticia para mandar notificación.
- Base de datos con id del juego, favorito y deseado.
- Sacar información de la noticia por la api de steam.
- Poder poner un juego en favoritos o deseados.
- Poder quitar un juego de favoritos o deseados.

Tecnologías y herramientas

Se ha usado Android studio como ide de desarrollo ya que android es el sistema operativo, que mayor presencia tiene y además movil es donde tiene más sentido el proyecto ya que es donde más accesibilidad para las notificaciones tiene el Usuario.

En el apartado librerías se ha usado Retrofit, Gson, Room, WorkManager y GLide.

Retrofit es la librería usada para obtener información de la API esta tiene un uso muy intuitivo y bien documentado, se ha elegido ya que al ser la más popular es muy fácil encontrar información.

Gson se usa para poder convertir en objetos los Json que son recuperados a través de la Api la elección de esta y no otra es por su gran compatibilidad con Retrofit y android en general al ser de Google.

Room permite convertir objetos a una base de datos de SQLite, esto permite un sencillo paso de los juegos de la API a la base de datos, además de que al ser una librería oficial de Android para interactuar con SQLite tiene mucha información al ser muy usada.

WorkManager nos permite dejar una tarea en segundo plano y ejecutarla en el momento que le digamos además de que si no es posible se pospone a otro momento.

El lenguaje del proyecto es Kotlin ya que es el lenguaje principal de android y la mayoría de la documentación está hecha en él. Aunque también se pueda usar Java.

Estimación de recursos y planificación:

	Abril				Mayo			
	SEMANA 1 (31-6)	SEMANA 2 (7-13)	SEMANA 3 (14-20)	SEMANA 4 (21-27)	SEMANA 1 (28-4)	SEMANA 2 (5-11)	SEMANA 3 (12-18)	SEMANA 4 (19-21) ENTREGA
Obtener datos del juego por la api de steam.								
Convertir datos en card.								
Poner filtros sencillos de usar en el buscador del usuario.								
Notificaciones al usuario.								
Saber cuando un juego está en oferta para mandar notificación.								
Saber cuando un juego tiene una noticia para mandar notificación.								
Base de datos con id del juego, favorito y deseado.								
Poder poner un juego en favoritos o deseados.								
Poder quitar un juego de favoritos o deseados.								
Sacar información de la noticia por la api.								
Construir noticias por los datos y listarlas.								

An lisis

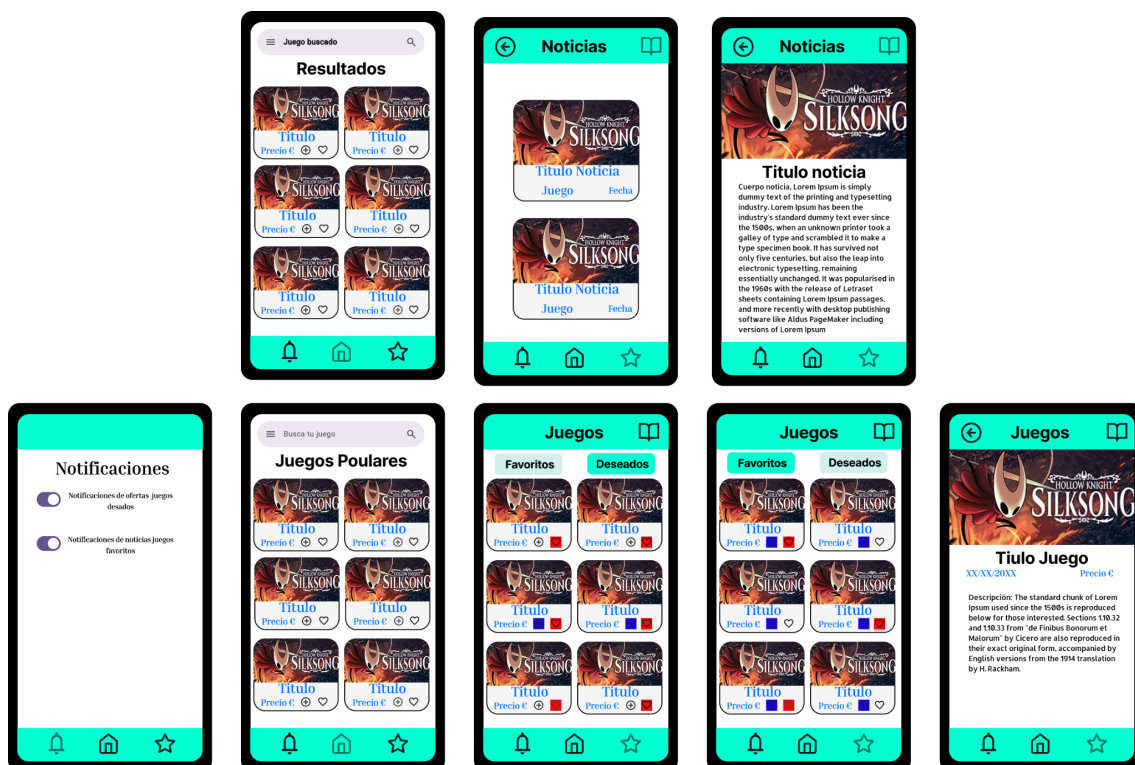
Requisitos funcionales:

- Buscar juegos de steam.
- Ver informaci n de estos juegos.
- Lista de favoritos y lista de deseados.
- Notificaciones de noticias de juegos favoritos.
- Notificaciones de ofertas de juegos deseados.
- Ver informaci n de noticias.
- Visualizar juegos populares.
- Visualizar tus listas de juegos.

Requisitos no funcionales:

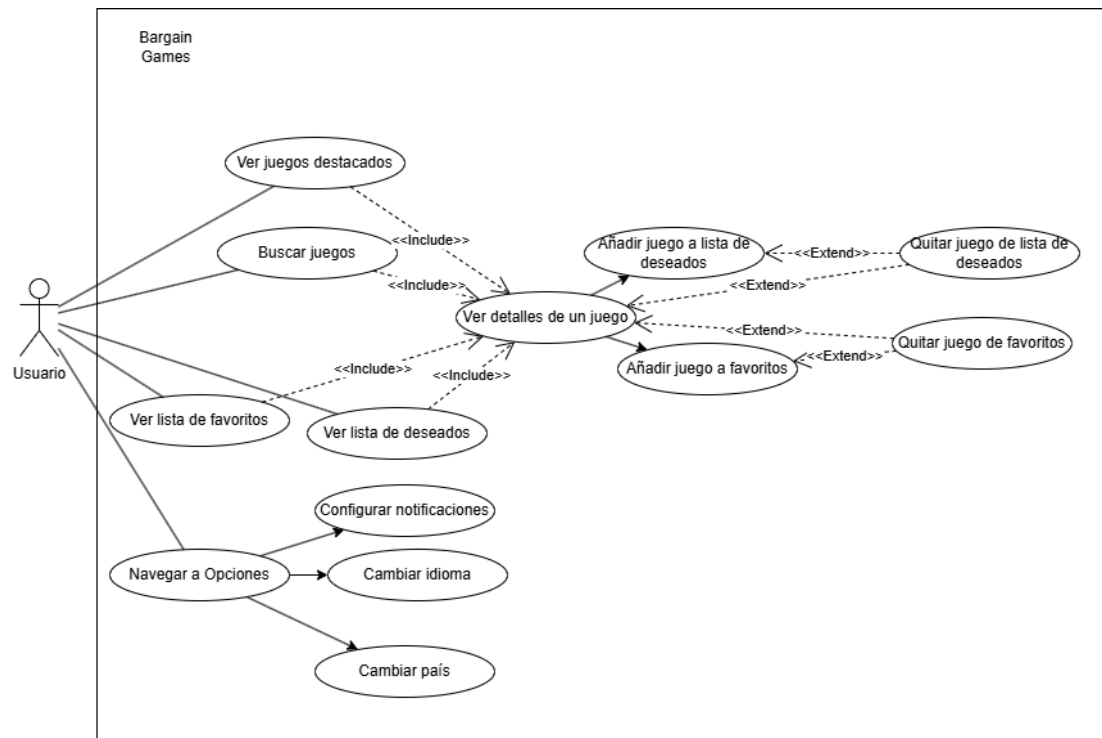
- Las notificaciones deben agruparse para evitar el env o masivo y no saturar al usuario.
- Las ofertas solo se revisar n de tus juegos en lista de deseados a las 10 am PDT (hora donde se actualizan las ofertas de steam).
- Las noticias se actualizan cada 6 horas y cuando el usuario ingresa a la app.
- Debe permitir la navegaci n fluida entre secciones sin necesidad de demasiados clics o acciones.
- La app debe ajustarse a distintos tama os de dispositivos.
- La app debe de ser intuitiva de usar.
- La app debe ajustarse al modo oscuro de los dispositivos.

Dise o



Diagramas

En anexos el diagrama de clases



Caso de uso 1	Ver juegos destacados
Alias	-
Actores	- Usuario
Requisito funcional	- Llamar a la API de steam y conseguir los juegos destacados del momento
Descripción	El usuario cuando ingrese a la APP o se mueva podrá ver los juegos destacados en steam en ese momento.
Referencias	Ver detalles de un juego (5)
Comentarios	Ningún comentario

Caso de uso 2	Buscar juegos
Alias	-
Actores	- Usuario
Requisito funcional	- Recuperar la palabra puesta por el usuario y llamar a la API de steam y conseguir los juegos que devuelvan según la palabra puesta

Descripción	El usuario podrá buscar cualquier juego de Steam a través de la búsqueda de este
Referencias	Ver detalles de un juego (5)
Comentarios	Ningún comentario

Caso de uso 3	Ver lista de deseados
Alias	-
Actores	- Usuario
Requisito funcional	- Recuperar los juegos que en la base de datos estén guardados como deseados
Descripción	El usuario podrá filtrar sus juegos guardados por deseados
Referencias	Ver detalles de un juego (5)
Comentarios	Ningún comentario

Caso de uso 4	Ver lista de favoritos
Alias	-
Actores	- Usuario
Requisito funcional	- Recuperar los juegos que en la base de datos estén guardados como favoritos
Descripción	El usuario podrá filtrar sus juegos guardados por favoritos
Referencias	Ver detalles de un juego (5)
Comentarios	Ningún comentario

Caso de uso 5	Ver detalles de un juego
Alias	-
Actores	- Usuario

Requisito funcional	- Obtener los datos del juego que el usuario quiera ver sus detalles a través de la API de steam
Descripción	El usuario podrá ver los detalles de cualquier juego que quiera
Referencias	Añadir juego a la lista de deseados(6)
Comentarios	Ningún comentario

Caso de uso 6	Añadir juego a la lista de deseados
Alias	-
Actores	- Usuario
Requisito funcional	- Se mete los datos del juego en la base de datos y se marca el campo deseado, si ya estaba en la base de datos solo se marca
Descripción	El usuario podrá marcar sus juegos como deseados y guardarlos
Referencias	Quitar juego a la lista de deseados (8)
Comentarios	Ningún comentario

Caso de uso 7	Añadir juego a la lista de favoritos
Alias	-
Actores	- Usuario
Requisito funcional	- Se mete los datos del juego en la base de datos y se marca el campo favoritos, si ya estaba en la base de datos solo se marca
Descripción	El usuario podrá marcar sus juegos como favoritos y guardarlos
Referencias	Quitar juego a la lista de favoritos(9)
Comentarios	Ningún comentario

Caso de uso 8	Quitar juego a la lista de deseados
----------------------	-------------------------------------

Alias	-
Actores	- Usuario
Requisito funcional	- Se quita la marca de la base de datos de deseados, si no hay otra se quita la base de datos
Descripción	El usuario podrá desmarcar sus juegos como deseados y quitarlos de guardados
Referencias	Ver detalles de un juego (5)
Comentarios	Ningún comentario

Caso de uso 9	Quitar juego a la lista de favoritos
Alias	-
Actores	- Usuario
Requisito funcional	- Se quita la marca de la base de datos de deseados, si no hay otra se quita la base de datos
Descripción	El usuario podrá desmarcar sus juegos como favoritos y quitarlos de guardados
Referencias	Ver detalles de un juego (5)
Comentarios	Ningún comentario

Caso de uso 10	Navegar menú de opciones
Alias	-
Actores	- Usuario
Requisito funcional	- Se mete a un menú
Descripción	El usuario podrá cambiar cosas en este menú
Referencias	Configurar notificaciones (11) Cambiar país(12) Cambiar idioma(13)

Comentarios	Ningún comentario
--------------------	-------------------

Caso de uso 11	Configurar notificaciones
Alias	-
Actores	- Usuario
Requisito funcional	- Se recibe las notificación según instrucciones del usuario
Descripción	El usuario podrá cambiar las notificaciones
Referencias	
Comentarios	Ningún comentario

Caso de uso 12	Cambiar país
Alias	-
Actores	- Usuario
Requisito funcional	- Se cambia el país con el que se hacen las llamadas a steam
Descripción	El usuario podrá cambiar su país
Referencias	
Comentarios	Ningún comentario

Caso de uso 13	Cambiar idioma
Alias	-
Actores	- Usuario
Requisito funcional	- Se cambia el idioma con el que se hacen las llamadas a steam
Descripción	El usuario podrá cambiar su idioma

Referencias	
Comentarios	Ningún comentario

Diagrama de entidad relación

Games
<u>PK id int NOT NULL</u>
name String
descripcion String
imagen String
capsuleImagen String
price String
favorito Boolean
deseado Boolean
free Boolean

Documentación

```
@GET(FETURED)
suspend fun getFeaturedCategories(
    @Query("l") language: String = "spanish",
    @Query("cc") countryCode: String = "ES"
): FeaturedCategories

@GET(SEARCH)
suspend fun getStoreSearch(
    @Query("term") term: String,
    @Query("l") language: String = "spanish",
    @Query("cc") countryCode: String = "ES"
): StoreSearch
```

Estas dos funciones que llaman a la API a través del uso de retrofit, se tiene que usar en una interfaz, usamos constantes para que si cambia la API solo tener que tocar el archivo constantes. Se usa "Query" para pasar un parámetro en la llamada en este caso el lenguaje y el país. En búsqueda pasamos el término que queremos buscar, es

el único que no tenemos iniciado de antes, el resto tienen un valor por defecto que al menos que se le pase otro parámetro será ese el que se le envíe a la API.

Con retrofit hay que crear una interfaz con las funciones que se quieran hacer con la API para luego declararlo así:

```
protected fun setupRetrofit() {  
    val retrofit = Retrofit.Builder()  
        .baseUrl(Constants.BASE_URL)  
        .addConverterFactory(GsonConverterFactory.create())  
        .build()  
    service = retrofit.create(GamesService::class.java)  
}
```

Con este método creamos la instancia de retrofit, aquí le asignamos en primero el URL base de la api que lo tenemos con las otras direcciones a la api una constante en otro archivo, luego le asignamos un convertidor de JSON en este caso Gson, y luego esa build de retrofit le damos la interfaz de las llamadas para asignarle los métodos que tiene que tener.

```
private fun buscarJuego(termino: String) {  
    lifecycleScope.launch(Dispatchers.IO) {  
        try {  
            val response = service.getStoreSearch(termino, language: "spanish", countryCode: "ES")  
            val games = response.games ?: emptyList()  
            withContext(Dispatchers.Main) {  
                if (games.isNotEmpty()) {  
                    searchAdapter.submitList(games)  
                    mostrarResultados()  
                } else {  
                    mostrarNoResultados()  
                }  
            }  
        } catch (e: Exception) {  
            Log.e(tag: "HomeFragment", msg: "Error buscando juego", e)  
            withContext(Dispatchers.Main) {  
                mostrarNoResultados()  
            }  
        }  
    }  
}
```

Esto es como se busca un juego, metemos el término el término se recoge de la barra de tareas que es propia de android y cuando se a buscar y esta algo escrito se manda aquí. Ahora mismo está el idioma por defecto en español y el país en España, esto va a ser editable en opciones en algún momento próximo. Dentro de la respuesta hay una lista "games", ahí se encuentran todos los juegos que corresponden a la búsqueda a

través de la API. Ahora hablamos con el layout y si la lista no está vacía metemos los juegos en el Layout a través de un adaptador y mostramos la lista de juegos, sino mostramos que no se han encontrado juegos.

```
@SerializedName("specials")
val specials: GameCategory?,

@SerializedName("coming_soon")
val comingSoon: GameCategory?,

@SerializedName("top_sellers")
val topSellers: GameCategory?,

@SerializedName("new_releases")
val newReleases: GameCategory?,
```

Los otros juegos importantes son recomendados, esto devuelve varias listas de juegos que de una categorías diferentes de steam.

```
@SerializedName("discount_percent")
var discountedPercent: Int,

@SerializedName("original_price")
val originalPrice: Int,

@SerializedName("final_price")
var finalPrice: Int,

val currency: String,
```

```
data class Price(
    val currency: String,
    val initial: Int,
    val final: Int,
    @SerializedName("discount_percent")
    val discountPercent: Int,
    @SerializedName("final_formatted")
    val finalFormatted : String
)
```

Por limitaciones de la Api de Steam se han tenido que hacer 3 modelos diferentes de objetos juego, uno para las búsquedas, otro para los juegos recomendados y un último para los detalles de juegos, cada uno tiene sus diferencias en cómo se llaman los valores y que valores tienen cada uno. El precio tanto en detalles como en búsqueda es un objeto aparte, mientras que en los juegos recomendados se tiene cada campo aparte.

```
interface GameItem {
    val id: Int
    val name: String
}
```

```
data class GameCategorized(
    override val id: Int,
    override val name: String,
    var discounted: Boolean,
```

Para facilitar el uso de estos en algunos métodos se creó una interfaz. Con los valores más importantes, aún así por como funciona Kotlin hay que hacer un overwrite en cada uno de los campos por mucho que estén en la interfaz ya que sino da un error en Kotlin.

Ahora vamos a hablar de las propias clases en sí, están declaradas como “data class”, esto sirve para que se generen tanto los getter como los setters de cada uno de las variables automáticamente, pero no solo hace eso, estas clases varias otras funciones que se declaran automáticamente, la otra que si usamos es equals esta función permite saber si dos clases del mismo tipo son iguales o no.

```
override fun onCreateView(  
    inflater: LayoutInflater,  
    container: ViewGroup?,  
    savedInstanceState: Bundle?  
): View {  
    _binding = FragmentHomeBinding.inflate(inflater, container, attachToParent: false)  
    return binding.root  
}
```

Para hacer más sencillo interactuar en las interfaces hemos usado en el código la función binding. Binding nos permite vincular nuestro xml de interfaz a un objeto con eso conseguimos que sea más legible y más sencillo interactuar con cualquier elemento de nuestra interfaz.

En la primera pantalla tenemos, que los juegos cambian según si buscamos un juego o no, la manera en la que se ha hecho es usar 2 “RecyclerView”, estos sirven para repetir un ítem cuantas veces quieras, se ha hecho un ítem juego con su nombre, imagen precio y 2 botones que cambian según si está en favorito o no.

```
private fun setupRecyclerView(){  
    binding.recyclerViewCategorias.apply {  
        layoutManager = StaggeredGridLayoutManager( spanCount: 2, RecyclerView.VERTICAL)  
        adapter = this@HomeFragment.listAdapter  
    }  
    binding.recyclerViewBusqueda.apply {  
        layoutManager = StaggeredGridLayoutManager( spanCount: 2, RecyclerView.VERTICAL)  
        adapter = this@HomeFragment.searchAdapter  
    }  
}
```

Para poder usar un recycled view hay que hacer un adaptador de un objeto a ese ítem, eso hace que como tenemos 3 objetos diferentes hay que hacer 3 adaptadores diferentes para cada uno.

```
override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): RecyclerView.ViewHolder {  
    context = parent.context  
    return ViewHolder(LayoutInflater.from(context).inflate(R.layout.item_game, parent, attachToRoot: false))  
}
```

Los adaptadores tiene varios métodos que necesitan ser puestos para que funcione, primero al crearlo le tenemos que dar un contexto es decir la Main Activity de donde viene y un layout que será el item que hemos hecho.

```
private class GameDiff : DiffUtil.ItemCallback<GameCategorized>() {  
    override fun areItemsTheSame(oldItem: GameCategorized, newItem: GameCategorized): Boolean {  
        return oldItem.id == newItem.id  
    }  
  
    override fun areContentsTheSame(oldItem: GameCategorized, newItem: GameCategorized): Boolean {  
        return oldItem == newItem  
    }  
}
```

En este método hace que cuando se cambie la lista del adaptador sepa si tiene que volver a renderizar el elemento o por el contrario el nuevo es el mismo al anterior. Para esto necesitamos que la clase sea data ya que tiene que comprobar si algo ha cambiado en el objeto.

```
override fun onBindViewHolder(holder: RecyclerView.ViewHolder, position: Int) {  
    val game = getItem(position)  
    (holder as ViewHolder).run {  
        setListener(game)  
        with(binding){  
            Nombre.text = game.name  
            Precio.text = "" + (game.finalPrice.toFloat() / 100) + game.currency  
  
            Glide.with(context)  
                .load(game.smallImage)  
                .diskCacheStrategy(DiskCacheStrategy.AUTOMATIC)  
                .into(imageGame)  
        }  
    }  
}
```

Este es el adaptador a partir de cada elemento que tenga una lista convierte el ítem en una interfaz. Glide es una librería para obtener imágenes de internet a través de una URL, usamos la imagen pequeña ya que carga más rápido y la resolución es más baja pero no se nota por el tamaño, está puesto para que la imagen sea guardada según el criterio de la librería.

Se ha asegurado de que si buscas un juego y luego vuelvas a la pestaña de búsqueda llegues a la misma búsqueda que estabas, de la siguiente manera:


```
override fun onResume() {  
    super.onResume()  
    if(search.isNotEmpty()){  
        buscarJuego(search)  
        mostrarResultados()  
    }else{  
        getGames()  
    }  
}
```

```
private fun mostrarResultados() {  
    binding.recyclerViewCategorias.visibility = View.GONE  
    binding.recyclerViewBusqueda.visibility = View.VISIBLE  
    binding.tvNoData.visibility = View.GONE  
}
```

Esto hace que si se reanuda el fragmento, se mire si hay algo en la barra de búsqueda si lo hay se busca y se mostrará solo las búsquedas para eso se utiliza View.Gone en cada elemento que no queremos que se muestre.

```
if (games.isNotEmpty()) {  
    mostrarCategorias()  
    listAdapter.submitList(list)  
}else{  
    mostrarNoResultados()  
}
```

También se comprueba si se han devuelto juegos sino se muestra un texto con que no se ha encontrado nada, esto para si el usuario tiene mala conexión a internet o ha hecho una búsqueda incorrecta lo pueda saber.

Para mirar los detalles del juego necesitamos darle una función onClick al adaptador ya que este por defecto no la tiene para eso creamos nuestra propia interfaz onClickListener.

```
interface OnClickListener {  
    fun onClick(gameItem: GameItem)
```

```
inner class ViewHolder(view: View) : RecyclerView.ViewHolder(view) {  
    val binding = ItemGameBinding.bind(view)  
  
    fun setListener(gameItem: GameItem) {  
        binding.root.setOnClickListener {  
            listener.onClick(gameItem)  
        }  
    }  
}
```

Y que se debe poder usar en cualquiera de los 3 adaptadores y se va hacer exactamente lo mismo usamos la interfaz game item y esta la instanciamos en cada uno de los adaptadores. Ahora hay que darle función al método onClick esto se hace en el fragmento.

```
override fun onClick(gameItem: GameItem) {  
    val fragment = GameDetailFragment.newInstance(gameItem.id)  
    val fragmentManager = requireActivity().supportFragmentManager  
    val transaction = fragmentManager.beginTransaction()  
  
    fragmentManager.fragments.forEach {  
        if (it.isVisible) transaction.hide(it)  
    }  
  
    transaction  
        .add(R.id.nav_host, fragment)  
        .addToBackStack(name: null)  
        .commit()  
}
```

Lo primero que hacemos es crear el fragment según el id del juego que hemos hecho click, luego llamamos al Fragmentmanager de la Activity principal y hacemos una transición, ocultamos todos los fragments, en principio solo hay uno visible pero se ocultan todos por que es más facil que sacar el que está visible y además esto evita cualquier tipo de error en la app, como nuestro fragment acaba de ser creado hay que instanciarlo en el contenedor de fragments aquí, al hacer esto tenemos nuestro nuevo fragment en pantalla.

```
companion object {  
    private const val ARG_APP_ID = "app_id"  
  
    fun newInstance(appId: Int): GameDetailFragment {  
        val fragment = GameDetailFragment()  
        val args = Bundle()  
        args.putInt(ARG_APP_ID, appId)  
        fragment.arguments = args  
        return fragment  
    }  
}
```

```
arguments?.let {  
    appId = it.getInt(ARG_APP_ID)  
}
```

Con este código creamos una instancia de cada fragment y pasamos el id del juego por Bundle y para sacar el id del bundle y aplicarlo en el fragmento en el método onCreate llamamos al bundle y sacamos la id y se lo asignamos a una variable del fragment. Con esto llamamos a la api y usamos el siguiente método.

```
@GET(APPDETAILS)
suspend fun getAppDetails(
    @Query("appids") appId: String,
    @Query("filters") filters: String = "",
    @Query("cc") country: String = "ES",
    @Query("l") language: String = "spanish"
): Map<Int, GameDetail>
```

Este método tiene un par de curiosidades, la primera es que aunque el id se un siempre un número se pueden pasar varios ids de varios juego, pero si se hace esto da error al menos que el filtro sea “price_overview” hay varios filtros disponible pero nosotros no usamos uno ya que el básico no devuelve el precio¹. Los otros 2 filtros son como siempre el idioma y el país, si por alguna razón la descripción o algún texto no está en el idioma seleccionado sale en inglés. La respuesta es un Mapa cuya clave es el número de id del juego y sus detalles.

```
lifecycleScope.launch(Dispatchers.IO) {
    try {
        val response = service.getAppDetails(appId.toString())
        val appDetails = response[appId]
        if (appDetails?.success == true && appDetails.data != null) {
            game = appDetails.data
            withContext(Dispatchers.Main) {
                binding.apply {
                    name.text = game.name
                    description.text = HtmlCompat.fromHtml(game.description, HtmlCompat.FROM_HTML_MODE_LEGACY)

                    context?.let {
                        Glide.with(it)
                            .load(game.image)
                            .diskCacheStrategy(DiskCacheStrategy.AUTOMATIC)
                            .into(imageGame)
                    }
                }
            }
        }
    }
}
```

Así ponemos las descripciones de html y las imágenes de internet en los fragmentos de detalles de juegos, como las descripciones están en html hay que decírselo al texto, pero eso no hace que puedas instanciar las imágenes que tienes dentro. El caché en automático hace que la librería decida si guardar en caché la imagen o no, si lo hace la próxima vez que se cargue esa imagen será mucho más rápido.

Para poder guardar un juego necesitamos una base de datos con Room podemos convertir nuestro objeto Game Detail en una tabla de una base de datos y luego recuperarlo automáticamente como un objeto.

```
@Entity(tableName = "Game")
data class GameData (
    @SerializedName("steam_appid")
    @PrimaryKey override val id: Int,

    @Database(entities = [GameData::class], version = 2)
    @TypeConverters(GameConverters::class)
    abstract class GameDatabase : RoomDatabase() {
        abstract fun gameDao(): GameDao
    }
}
```

Al objeto se le pone una anotación de entidad y se le puede asignar un nombre a la tabla si no se pone nada será el nombre del objeto, hay que asignarle una Primary Key y el resto de atributos serán las columnas de la base de datos según el nombre que tenga declarado.

Para instanciar la base de datos hay que crear una clase abstracta donde hay que definir la entidad y la versión, si se cambia la base de datos y se quiere que se actualice la base de datos para no perder nada, hay que hacer también una emigración de base de datos, que se verá después. Luego tenemos el Converter esto sirve para poder pasar objetos a una columna de la base de datos.

```
class GameConverters {
    @TypeConverter
    fun fromJsonStr(value: String?): Price? {
        return value?.let { Gson().fromJson(it, Price::class.java) }
    }

    @TypeConverter
    fun fromRating(value: Price?): String? {
        return value?.let { Gson().toJson(it) }
    }
}
```

El converter simplemente transforma el objeto a JSON para poder meterlo en una columna de la base de datos y luego cuando se recupera como objeto lo pasa de JSON a objeto, esto se hace porque como el precio es un objeto y está dentro de los detalles del juego no tiene sentido hacer una tabla aparte.

```
class GameApplication : Application() {
    companion object {
        lateinit var database: GameDatabase
    }
}
```

```
<application
    android:name=".model.database.GameApplication"
    android:allowBackup="true"
```

La clase donde se inicia tiene que ser hija de application ya que esto le permite

iniciarse una sola vez automáticamente cuando la aplicación se inicie, se pueda acceder a esta desde cualquier parte de la app y siempre está disponible mientras lo esté la app. Para que funcione esto se debe añadir el nombre de la clase en el manifest de la aplicación.

```
database = Room.databaseBuilder(context: this,
    GameDatabase::class.java,
    name: "GameDatabase")
    .addMigrations(MIGRATION_1_2)
    .build()
```

La base de datos se inicia de esta manera en el onCreate de la clase, le pasamos un contexto, la clase abstracta hecha anteriormente y el nombre del archivo, aquí también se ponen las migraciones de la base de datos, después de esto se ejecuta la base de datos.

```
val MIGRATION_1_2 = object : Migration(1,2){
    override fun migrate(db: SupportSQLiteDatabase) {
        db.execSQL("sql: \"ALTER TABLE Game ADD COLUMN favorito INTEGER DEFAULT 0 NOT NULL\")
        db.execSQL("sql: \"ALTER TABLE Game ADD COLUMN deseado INTEGER DEFAULT 0 NOT NULL\")
    }
}
```

En las migraciones se cambian los datos de la base de datos antigua a la nueva, hay muchas formas de que pase esto, pero principalmente son el añadido o cambio columnas a la base de datos, cuando se crea una nueva columna se debe indicar siempre el nuevo dato correspondiente por defecto, aunque con where se puede asignar dependiendo de los otros valores de la base de datos. Las migraciones las nombramos con detrás el número de la versión inicial y después el de la versión final y antes de esto se le llama migration.

```
@Dao
interface GameDao {
    @Query("SELECT * FROM Game")
    fun getAllGame(): MutableList<GameData>

    @Insert
    fun addGame(game: GameData): Long

    @Delete
    fun deleteGame(game: GameData)
```

Para hacer acciones en la base de datos usamos una interfaz, DAO, para indicar a

retrofit los métodos y sus acciones, en Query podemos ejecutar cualquier sentencia aunque para las updates es mejor usar Update en el cual se pasa el nuevo objeto y lo edita en la base de datos dejándolo igual al que hemos pasado. Insert y delete funcionan igual excepto que insert inserta el nuevo objeto en la base de datos y delete lo elimina. El retorno pueden ser listas del objeto o el objeto.

```
}else{
    game.favorito = true
    val result = GameApplication.database.gameDao().addGame(game)
    gameDB = game
    if (result != -1L) {
        withContext(Dispatchers.Main) {
            Snackbar.make(view, text: "Juego añadido a favoritos", Snackbar.LENGTH_SHORT)
                .show()
            binding.apply {
                cbFavorite.setImageResource(R.drawable.favorite_24)
            }
        }
    }
}
```

Así añadimos cuando le damos al botón de añadir a favoritos un juego en la base de datos y también los instanciamos como un objeto y lo abismos al usuario a través de un sackbar y cambiamos la imagen del botón, se hace lo mismo para deseados. Para que inicie con el botón cambiado cada vez que nos metemos en el juego, se llama a la base de datos para recibir si el juego está en la base de datos o no.

```
lifecycleScope.launch(Dispatchers.IO) {
    if(gameDB != null) {
        if (gameDB!!.favorito){
            gameDB!!.favorito = false
            GameApplication.database.gameDao().updateFavorito(gameDB!!)
            withContext(Dispatchers.Main) {
                Snackbar.make(view, text: "Juego quitado de favoritos", Snackbar.LENGTH_SHORT)
                    .show()
                binding.apply {
                    cbFavorite.setImageResource(R.drawable.favorite_no_24)
                }
            }
        }
        if(!gameDB!!.deseado){
            GameApplication.database.gameDao().deleteGame(game)
        }
    }
}
```

Luego si se toca el botón teniendo el juego una entrada en la base de datos se mirara si el juego es favorito o no y se pone el contrario y si se quita de favoritos se va a mirar que el juego no esté en deseados, si no está como señalado en base de datos se elimina de ella.

```
CoroutineScope(Dispatchers.IO).launch {
    val gameDb = GameApplication.database.gameDao().getGame(gameStore.id)
    withContext(Dispatchers.Main) {
        setListener(gameStore)
        with(binding) {
            if (gameDb?.favorito == true) {
                cbFavorite.setImageResource(R.drawable.favorite_24)
            } else {
                cbFavorite.setImageResource(R.drawable.favorite_no_24)
            }
            if (gameDb?.deseado == true) {
                cbDeseado.setImageResource(R.drawable.add_circle_24)
            } else {
                cbDeseado.setImageResource(R.drawable._add_circle_no_24)
            }
        }
    }
}
```

Para que cuando salga un juego podamos ver si está en la base de datos como favorito o deseado usamos este código, usamos una corrutina para poder llamar a la base de datos ya que esto no se puede hacer en el hilo principal, porque Android necesita que la interfaz sea fluida y responsiva y si se hace una operación a la base de datos en el hilo principal de la creación de la interfaz dará un error por la pérdida de fluidez.

Luego usamos withContext esto nos devuelve al hilo principal de la ejecución de la interfaz, ahí iniciamos el Listener que es el onClick y con blinding ponemos los iconos de los juegos. También se puede ver en el código como no solo ponemos el icono de si está añadido también el de que no está añadido, esto es necesario ya que si no lo cambiamos cuando vayamos moviendo el recycle view no se actualizará los iconos, ya que hay este elemento va reciclando los ítems.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/navigation_home"
        android:icon="@drawable/home"
        android:title="Home"/>
    <item
        android:id="@+id/navigation_favorite"
        android:icon="@drawable/favorite_24"
        android:title="Guardados"/>
</menu>
```

Con esto creamos los botones del menú de abajo de la aplicación le ponemos un

icono y un nombre, el nombre está guardado en el xml de strings. Ponemos un id para poder referirnos desde el código a cada botón.

```
private fun setupBottomNav() {  
    val homeFragment = HomeFragment()  
    val favouriteFragment = FavouriteFragment()  
  
    with(supportFragmentManager){  
        beginTransaction()  
        .add(R.id.nav_host, homeFragment, tag: "home")  
        .add(R.id.nav_host, favouriteFragment, tag: "favourite")  
        .hide(favouriteFragment)  
        .commit()  
    }  
}
```

Asignamos cada uno de los fragmentos a la barra de navegación y ocultamos el de la pestaña de guardados para mostrar la principal, a cada uno de las pestañas le añadimos un tag.

```
binding.navView.setOnItemSelectedListener { menuItem ->  
  
    val fragmentToShow : Fragment = if(menuItem.itemId == R.id.navigation_home){  
        homeFragment  
    }else{  
        favouriteFragment  
    }  
  
    beginTransaction().apply {  
        fragments.forEach { hide(it) }  
        show(fragmentToShow)  
        commit()  
    }  
    true  
}
```

Ahora para cambiar entre los fragmentos usamos este método según el icono que se seleccione. Luego ocultamos todos los fragmentos, está hecho para que si no es uno de los dos sigan funcionando y se oculte. Y enseñamos el fragmento que vamos a enseñar. Al final se devuelve un "True" para que salga seleccionado el icono.


```
private fun getGames() {  
    lifecycleScope.launch(Dispatchers.IO) {  
        try {  
            val games = GameApplication.database.gameDao().getAllGame()  
            withContext(Dispatchers.Main) {  
                mostrarResultados()  
                detailAdapter.submitList(games)  
            }  
        } catch (e: Exception) {  
            Log.e(tag: "MainActivity", msg: "Error al obtener juegos", e)  
            withContext(Dispatchers.Main) {  
                mostrarNoResultados()  
            }  
        }  
    }  
}
```

Para sacar los juegos guardados en la base de datos llamamos al m todo getAllGames que devuelve una lista de todos los juegos en la base de datos, como siempre salimos del corrutina al hilo de la interfaz, mostramos los juegos y metemos la lista en el adaptador de detalles de juegos. El try chat se encarga de si hay un error mostrar al usuario que ha habido un error recogiendo los juegos de la base de datos.

```
binding.buttonFavorite.setOnClickListener{  
    if(favorite){  
        getGames()  
        favorite = false  
    }else{  
        getFavoritesGames()  
        favorite = true  
        wish = false  
    }  
}
```

```
binding.buttonDeseados.setOnClickListener{  
    if(wish){  
        getGames()  
        wish = false  
    }else{  
        getWishGames()  
        wish = true  
        favorite = false  
    }  
}
```

Con estos dos m todos vamos cambiando los juego que sean filtrados por deseados o favoritos, si le das a un bot n mientras est  activo se sacar n los juego sin filtros, para sacar los juego filtrados se usa unos m todos casi iguales a getGames pero con a la base de datos se le dice que devuelva solo los juegos favoritos o deseados respectivamente.

```
override fun onResume() {  
    super.onResume()  
    if(favorite){  
        getFavoritesGames()  
    }else if (wish){  
        getWishGames()  
    }else{  
        getGames()  
    }  
}
```

En esta interfaz como en la otra cuando se reanuda el fragmento, se cambia entre los 3 tipos de búsquedas según lo que haya seleccionado el usuario. Actualmente como los fragmentos no se paran si no se ocultan en la mayoría de situaciones no llega a ejecutarse.

```
private fun notificationChanel() {  
    val channel = NotificationChannel(  
        id: "bargainGames",  
        name: "Ofertas de Steam",  
        NotificationManager.IMPORTANCE_HIGH  
    ).apply {  
        description = "Notificaciones de ofertas de juegos deseados"  
    }  
  
    val manager = this.getSystemService(NotificationManager::class.java)  
    manager.createNotificationChannel(channel)  
}
```

Este código crea un canal para las notificaciones de la app, las notificaciones necesitan un canal a partir de la Api 26 de android que justamente es la Api mínima que se ha puesto por un problema con el icono de la app, si se quisiera bajar más la versión de la api habría que poner un if buscando la versión señalando que deber ser menor a la versión O, Oreo, para no que no se cree ya que antes de esto funcionaban sin canal.

```
private fun workerDiario() {  
    val constraints = Constraints.Builder()  
        .setRequiredNetworkType(NetworkType.CONNECTED)  
        .build()  
  
    val dailyRequest = PeriodicWorkRequestBuilder<OfertasWorker>(  
        repeatInterval: 1, TimeUnit.DAYS  
    ).setInitialDelay(calcularHora(), TimeUnit.MILLISECONDS)  
        .setBackoffCriteria(  
            BackoffPolicy.EXPONENTIAL,  
            WorkRequest.DEFAULT_BACKOFF_DELAY_MILLIS,  
            TimeUnit.MILLISECONDS  
        ).setConstraints(constraints)  
        .build()  
}
```

Para crear una notificaci3n diaria primero tenemos que crear un subproceso que se ejecute cada d́a, para eso le indicamos que el proceso se repite todos los d́as y calculamos la diferencia de tiempo actual con la diferencia de tiempo restante para la hora que se desea, las 6 de la tarde.

Despu3s de insertar la diferencia de tiempo hay que decir cuando volver a ejecutar si la el proceso falla, el primero es como incrementa el tiempo de cada nuevo intento de ejecutar la tarea, esta instrucci3n hace que el tiempo entre cada una se de exponencial, se hace aś ya que esto consume menos recursos, aunque tambi3n est3 la opci3n de que los tiempos sean lineales.

Luego es cuanto tiempo se quiere entre un intento de ejecuci3n y otro la medida de este tiempo, se est3 usando el tiempo recomendado en android actualmente ya que se supone que es el m3s optimizado para esta versi3n, hay que tener en cuenta que el m3ximo retraso posible es de 5 horas cuando se llegue ah́ el siguiente tambi3n ser3 de 5 horas, llegar a este punto se tarda sobre 10 horas y media, en ese momento se habr3n habido 12 intentos de notificaciones.

Finalmente metemos las constraints estas hacen que la tarea solo se ejecute cuando haya una situaci3n espećfica, en este caso cuando haya conexi3n a la red, ya que al ser una descarga muy pequéa, solo son los datos del precio no es ning3n problema, que se ejecute incluso con datos.

```
private fun calcularHora(): Long {  
    val now = Calendar.getInstance()  
    val target = Calendar.getInstance().apply {  
        set(Calendar.HOUR_OF_DAY, 11)  
        set(Calendar.MINUTE, 0)  
        if (before(now)) {  
            add(Calendar.DAY_OF_YEAR, amount: 1)  
        }  
    }  
    return target.timeInMillis - now.timeInMillis  
}
```

El c3lculo de tiempo lo hacemos con Calendar y usamos el m3todo getInstance para coger el momento de ahora y hacemos otro que sea este momento y ponemos tanto el minuto como la hora en las 18:00. Luego restamos el tiempo que queremos al actual y tenemos la diferencia del tiempo en milisegundos que tiene que esperar para realizar la tarea.

```
WorkManager.getInstance(context: this).enqueueUniquePeriodicWork(  
    uniqueWorkName: "ofertas_diarias",  
    ExistingPeriodicWorkPolicy.KEEP,  
    dailyRequest  
)
```

Finalmente para programar la tarea hay que hacer una instancia de ella, primero tenemos el nombre único de la tarea, es un id que tiene la tarea en forma de nombre. Luego está una constante del sistema, está dictamina que hay que hacer si ya hay otro Worker en funcionamiento, en nuestro caso hemos dejado el anterior es decir keep, esto lo hacemos ya siempre que se inicia la app se inicia el worker y si cambiamos el worker cada vez que pase esto consumiría más recursos de los necesarios. Y luego metemos finalmente nuestra tarea asignada.

```
class OfertasWorker(  
    appContext: Context,  
    workerParams: WorkerParameters  
) : CoroutineWorker(appContext, workerParams) {  
  
    override suspend fun doWork(): Result {  
        val context = applicationContext  
    }  
}
```

Creamos una clase de CoroutineWorker a éste hay que pasarle pasa automáticamente un contexto y los parámetros asignados a las tarea, después sobrescribimos el método de hacer una tarea en este método sacamos el contexto con el el método getApplicationContext como es un objeto de kotlin la llamada se escribe como la del objeto en sí.

Ahora sacamos cada uno de los juegos de la base de datos en deseados, luego metemos cada id en un string con una coma entre medio de ellas. Luego instanciamos retrofit para la API de steam y usamos el método de gameDetails con el filtro de price y como ids pasamos el string con todos los ids. Esto nos devuelve un map con clave el id y los detalles de juego como información.

```
val juegosEnOferta = mutableListOf<String>()
for (juego in juegosDeseados) {
    val data = response[juego.id]?.data

    if (data?.price != null && data.price != juego.price) {
        if (data.price!!.discountPercent > juego.price!!.discountPercent) {
            juegosEnOferta.add("${juego.name} - ${data.price!!.discountPercent}% - ${data.price!!.finalFormatted}")
        }
        juego.price = data.price
        GameApplication.database.gameDao().update(juego)
    }
}
```

En la lista de juegos con ofertas vamos a meter un String por cada juego que est  en una nueva oferta, para esto miramos el precio, comprobamos que no sea nulo por comodidad ya que si es nulo el precio da un error, por eso si el juego no tiene precio en la base de datos, no se manda el id ya que dar a un error.

Luego comprobamos si el objeto precio es diferente el nuevo al viejo. Si esto pasa vamos a comprobar si hay un descuento, ya que puede ser que se haya quitado el descuento anterior, si el descuento es mayor al anterior guardamos un string que diga el nombre del juego, el precio final y el porcentaje de descuento. Luego simplemente actualizamos los datos del precio del juego y lo actualizamos en la base de datos.

```
if (juegosEnOferta.isNotEmpty()) {
    mostrarNotificacion(context, juegosEnOferta.joinToString(separator: "\n"))
}

return Result.success()
```

Comprobamos si los juegos en ofertas est  vac o y si no est  vac o llamamos a la notificaci n a la que le pasamos el contexto y juntamos la lista en un string en el que cada elemento est  separado por un salto de l nea.

Finalmente pasamos que el worker se ha ejecutado con  xito, si pasamos en alg n lugar que no ha resultado con  xito se repetir  seg n lo que le hemos pasado hasta que resulte.

```
private fun mostrarNotificacion(context: Context, texto: String) {
    val intent = Intent(context, MainActivity::class.java).apply {
        flags = Intent.FLAG_ACTIVITY_NEW_TASK or Intent.FLAG_ACTIVITY_CLEAR_TASK
    }

    val pendingIntent = PendingIntent.getActivity(
        context,
        requestCode: 0,
        intent,
        PendingIntent.FLAG_IMMUTABLE
    )
}
```

Para hacer que la notificaci n te mande, cuando le des click, a la main activity

necesitamos hacer un intent. Dentro del intent hacemos un flag que nos permite iniciar una nueva actividad eliminando lo que estuviera iniciado anteriormente. Esto evita que el usuario pueda volver sin querer a la pestaña anterior o se le que iniciada 2 veces la misma actividad.

Para poder meterlo en la notificación tenemos que convertirlo en un PendingIntent, le tenemos que pasar el contexto, un código privado de la notificación, el intent instanciado anteriormente y si el PendingIntent que va a ser modificable o no.

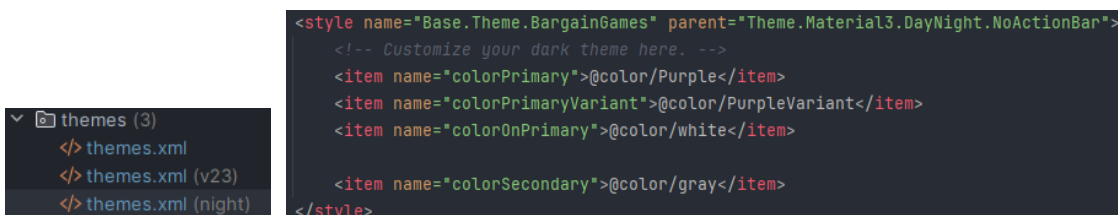
```
val notification = NotificationCompat.Builder(context, channelId: "bargainGames")
    .setSmallIcon(R.drawable.icon_games)
    .setContentTitle("¡Juegos en oferta!")
    .setContentText("Juegos de tu lista de deseados están en oferta")
    .setStyle(NotificationCompat.BigTextStyle().bigText(texto))
    .setContentIntent(pendingIntent)
    .build()

val manager = context.getSystemService(Context.NOTIFICATION_SERVICE) as NotificationManager
manager.notify(id: 1, notification)
```

Y finalmente creamos la notificación, hay que pasarle un contexto y el id del canal creado anteriormente. Luego le ponemos un icono el que se quiera no, se va aplicar el color de ese icono ya que Android pone uno por defecto, también ponemos el título y el texto de la notificación.

Después añadimos un texto grande, esto es el texto que sale cuando das a la notificación para ver más detalles de la notificación. Le metemos el PendingIntent para cuando le demos click a la notificación nos meta en la actividad. Luego hacemos build a la notificación.

Luego tenemos que llamar al manager del sistema de Android para tener los servicios de notificación del sistema. Luego le damos un id y pasamos la notificación si cuando se cree la notificación hay otra con el mismo id se eliminará la anterior y se actualizará la nueva.



The image shows a screenshot of an IDE. On the left, there is a file explorer showing a folder named 'themes' containing three files: 'themes.xml', 'themes.xml (v23)', and 'themes.xml (night)'. On the right, the content of 'themes.xml' is displayed, showing a custom theme named 'Base.Theme.BargainGames' that inherits from 'Theme.Material3.DayNight.NoActionBar'. The theme includes several color items: 'colorPrimary' (purple), 'colorPrimaryVariant' (purple variant), 'colorOnPrimary' (white), and 'colorSecondary' (gray).

```
<style name="Base.Theme.BargainGames" parent="Theme.Material3.DayNight.NoActionBar">
    <!-- Customize your dark theme here. -->
    <item name="colorPrimary">@color/Purple</item>
    <item name="colorPrimaryVariant">@color/PurpleVariant</item>
    <item name="colorOnPrimary">@color/white</item>
    <item name="colorSecondary">@color/gray</item>
</style>
```

Para cambiar el tema de la aplicación de android hay que hacer 3 temas en diferentes carpetas. El primero está en valores y son los colores de la aplicación en normal, el segundo está en la carpeta v23 este son cosas que no soportan los dispositivos inferiores a la api-23 y el tercero es el cuando el sistema está en modo oscuro.

Simplemente cambiamos los colores según los ítems como el color primario. En el V-23 se cambian cosas de la interfaz de android como la barra de abajo de la app.

Pruebas

Se ha hecho algún test muy sencillo en la app, y actualmente como el usuario solo puede meter por el buscador hay pocos valores máximos. Y se han comprobado principalmente en la entrada de API aunque no debería haber valores incorrectos.

Datos a meter	Dato esperado
Porcentaje descuento	Menor a 100 mayor a 0
Precio final	Menor a precio inicial o igual

Conclusiones

Se ha completado la parte más importantes, con todas sus dificultades, la parte de los juego ha sido un gran desafío ya que cada vez que recibía los datos venían de una forma diferente, entonces tuve que buscar una manera de poder usar la misma interfaz para mostrar 3 tipos de objetos diferentes.

Se ha tenido dificultades con los juegos que no tienen precio ya que estos pueden ser por ser gratis o que no hayan salido, en notificación me daba un crash por estos juegos ya que la api de steam no devuelve correctamente el valor. Por esto se ha hecho que estos juegos por ahora no se busquen su precio.

Pero aun así se han conseguido hacer tanto las notificaciones como todo el tema de los juegos. Además de guardarlos en una base de datos local, donde guardadas tus juegos en dos categorías favoritos y deseados. No se ha logrado meter las notificaciones ya que son una llamada a la api distinta y con una estructura distinta y el tiempo pensado para eso se lo han llevado los otros 2 objetos de los juegos. Lo otro que tampoco se ha conseguido es la pestaña de configuración ya que no se ha encontrado mucha información del tema.

Siento una gran satisfacción en general con el proyecto aunque hayan faltado parte de los objetivos, ya que se ha aprendido mucho de cómo funciona Android studio en general ya que se han tocado la mayoría de funciones de este IDE. He visto bastantes librerías muy usadas en estos tipos de proyectos y he aprendido cómo usarlas.

También el funcionamiento de los hilos de android y cuando y como tienes que pasar de hilo para realizar la actividad que quieres hacer.

También he podido entender cómo hacer un menú de una manera mucho más sencilla de la que pensé que podría ser, y se ha aprendido cómo cambiar de interfaces de varias maneras con los fragmentos. Me ha parecido muy importante aprender hacer notificaciones en esta tecnología y saber cómo programarlas, además de programarlas de una manera que sea compatible con sistemas más antiguos.

Vías futuras

Una de las próximas cosas que me gustaría añadir sería una pestaña, en forma de activity, donde puedas revisar bien todas las notificaciones que hayan salido, haciendo que esta sea donde te lleven las notificación cuando le des click. Eso se suma a un apartado de noticias donde puedas ver todas las noticias, de tus juegos favoritos con sus notificaciones y una página para leer correctamente estas.

También un menú de configuración donde puedas cambiar tu moneda, país e idioma y que desde ahí se configuren las notificaciones y desde ahí instanciar las notificaciones para que no se inicien cada vez que entras en la app.

Glosario

Api: Interfaz de Programación de Aplicaciones es el conjunto de protocolos o reglas que permiten a las aplicaciones conectarse. Esto se aplica para steam para recibir datos y también para las diferentes versiones de android.

Fragment: vistas reutilizables de android que se pueden instanciar de varias maneras y cambiar rápidamente, este tiene que estar por encima de una activity

Activity: vista principales de la aplicación principal que en la que tienen que instanciar cada elemento.

IDE: Entorno de Desarrollo Integrado, es la interfaz de programación.

Xml: Lenguaje de Marcado Extensible. Es una manera de expresar la información dentro de varios sistemas incluido android studio.

Bibliograf́a/Webgraf́a

1. RJackson. (n.d.). *User:RJackson/StorefrontAPI*. Team Fortress Wiki. Recuperado el 21 de mayo de 2025, de <https://wiki.teamfortress.com/wiki/User:RJackson/StorefrontAPI>
2. SteamDB. (2023, 28 de julio). *Store Prices API*. Steam Database Blog. <https://steamdb.info/blog/store-prices-api/>
3. Revadike. (n.d.). *Get App Details*. Internal Steam Web API Wiki. GitHub. Recuperado el 21 de mayo de 2025, de <https://github.com/Revadike/InternalSteamWebAPI/wiki/Get-App-Details>
4. Android Developers. (s.f.). *Migrar versiones de bases de datos*. Android Developers. Recuperado el 21 de mayo de 2025, de <https://developer.android.com/training/data-storage/room/migrating-db-versions?hl=es-419>
5. Android Developers. (s.f.). *Trabajos persistentes con Worker*. Android Developers. Recuperado el 21 de mayo de 2025, de <https://developer.android.com/develop/background-work/background-tasks/persistent/treading/worker?hl=es-419>
6. App Dev Insights. (2023, 3 de octubre). *Work Manager Android: Reliable background processing*. Medium. <https://medium.com/@appdevinsights/work-manager-android-6ea8daad56ee>

ANEXOS

1 Diagrama de clases

