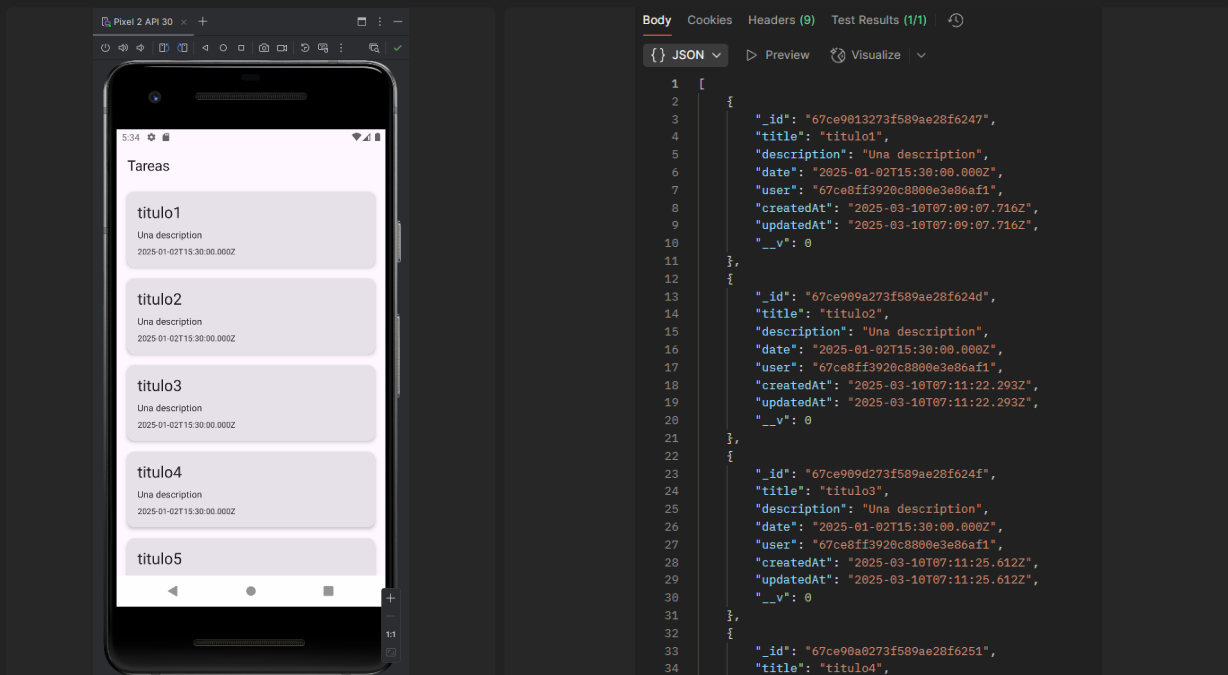


## Tarea 3.- Desarrollo de una Aplicación Móvil Nativa con Consumo de API REST

**Lopez Cardoza Victor Josue**

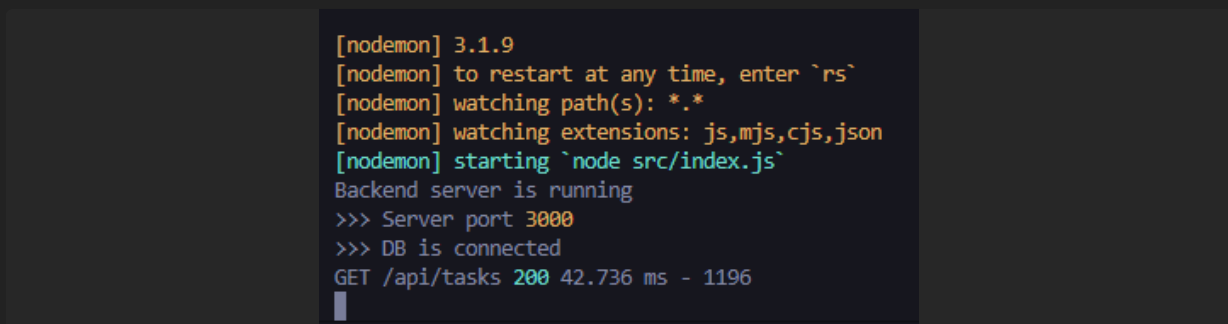
### Ejercicio 1: Creación de un Backend Básico y Conexión con Android

En la siguiente imagen se muestra la respuesta del backend, la información recibida es un arreglo de objetos JSON, que están listos para ser procesados por la aplicación móvil. También podemos ver que dentro del servidor se recibió correctamente la petición get para obtener las tareas.



The image shows two side-by-side screenshots. The left screenshot displays an Android application interface with a list titled "Tareas" containing five items: "titulo1", "titulo2", "titulo3", "titulo4", and "titulo5". Each item has a description "Una description" and a timestamp "2025-01-02T15:30:00.000Z". The right screenshot shows a REST client interface with the "Body" tab selected, displaying a JSON array of five task objects. The JSON structure is as follows:

```
1 [
2   {
3     "_id": "67ce9013273f589ae28f6247",
4     "title": "titulo1",
5     "description": "Una description",
6     "date": "2025-01-02T15:30:00.000Z",
7     "user": "67ce8ff3920c8800e3e86af1",
8     "createdAt": "2025-03-10T07:09:07.716Z",
9     "updatedAt": "2025-03-10T07:09:07.716Z",
10    "__v": 0
11  },
12  {
13    "_id": "67ce909a273f589ae28f624d",
14    "title": "titulo2",
15    "description": "Una description",
16    "date": "2025-01-02T15:30:00.000Z",
17    "user": "67ce8ff3920c8800e3e86af1",
18    "createdAt": "2025-03-10T07:11:22.293Z",
19    "updatedAt": "2025-03-10T07:11:22.293Z",
20    "__v": 0
21  },
22  {
23    "_id": "67ce909d273f589ae28f624f",
24    "title": "titulo3",
25    "description": "Una description",
26    "date": "2025-01-02T15:30:00.000Z",
27    "user": "67ce8ff3920c8800e3e86af1",
28    "createdAt": "2025-03-10T07:11:25.612Z",
29    "updatedAt": "2025-03-10T07:11:25.612Z",
30    "__v": 0
31  },
32  {
33    "_id": "67ce90a0273f589ae28f6251",
34    "title": "titulo4",
```



The terminal screenshot shows the following output:

```
[nodemon] 3.1.9
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,cjs,json
[nodemon] starting `node src/index.js`
Backend server is running
>>> Server port 3000
>>> DB is connected
GET /api/tasks 200 42.736 ms - 1196
```

### Código fuente de la aplicación

#### MainActivity (Punto de entrada)

La clase `MainActivity` extiende `ComponentActivity`, lo que significa que es la actividad principal de la aplicación. En el método `onCreate`, se establecen las configuraciones iniciales:

1. **enableEdgeToEdge()**: Esta función permite que la interfaz ocupe toda la pantalla, eliminando los márgenes de la barra de estado y la barra de navegación.
2. **setContent**: Aquí se define el contenido de la interfaz utilizando Jetpack Compose.
3. **TasksTheme**: Aplica un tema personalizado a la aplicación.
4. **Scaffold**: Es un contenedor estructural que proporciona una plantilla para la pantalla, incluyendo elementos como una barra superior ( `TopAppBar` ).

Ejemplo del `Scaffold` en el código:

```
Scaffold(
    modifier = Modifier.fillMaxSize(),
    topBar = {
        TopAppBar(
            title = { Text("Tareas") },
            colors = TopAppBarDefaults.topAppBarColors()
        )
    }
) { innerPadding ->
    TaskListScreenWrapper(modifier = Modifier.padding(innerPadding))
}
```

Aquí se está creando una barra superior con el título "Tareas" y luego se llama a `TaskListScreenWrapper`, pasando un `Modifier.padding(innerPadding)` para que respete los márgenes internos.

## **TaskListScreenWrapper (Encapsulación y Configuración de Retrofit)**

Esta función `@Composable` se encarga de configurar la conexión con la API y de instanciar el `ViewModel` que manejará la lógica de la pantalla de lista de tareas.

### 1. **Interceptor de Logs**

Se usa `HttpLoggingInterceptor` para registrar todas las solicitudes y respuestas HTTP. Esto es útil para depuración:

```
val loggingInterceptor = HttpLoggingInterceptor().apply {
    level = HttpLoggingInterceptor.Level.BODY
}
```

Configurado en `BODY`, lo que significa que imprimirá el contenido completo de las respuestas y solicitudes.

### 2. **Cliente HTTP ( `OkHttpClient` )**


Se crea un cliente HTTP personalizado que incluye el interceptor de logs:

```
val okHttpClient = OkHttpClient.Builder()
    .addInterceptor(loggingInterceptor)
    .build()
```

Esto permitirá ver en la consola todas las peticiones que se realicen a la API.

### 3. Retrofit (Conexión a la API)

Se configura Retrofit para comunicarse con la API en `http://10.0.2.2:3000/api/`.

 **Nota:** `10.0.2.2` es la dirección IP especial que permite que el emulador de Android se comunique con el servidor que corre en la máquina local.

```
val retrofit = Retrofit.Builder()
    .baseUrl("http://10.0.2.2:3000/api/")
    .addConverterFactory(GsonConverterFactory.create())
    .client(okHttpClient)
    .build()
```

- `baseUrl` : Define la URL base de la API.
- `addConverterFactory(GsonConverterFactory.create())` : Convierte automáticamente las respuestas JSON en objetos Kotlin.

### 4. Instanciación del Repositorio y ViewModel

```
val api = retrofit.create(TaskApi::class.java)
val repository = TaskRepository(api)
val viewModel: TaskViewModel = viewModel(factory =
TaskViewModel.Factory(repository))
```

- `TaskApi` : Interfaz donde se definen los métodos para interactuar con la API.
- `TaskRepository` : Capa que maneja la obtención de datos.
- `TaskViewModel` : Maneja la lógica de negocio y el estado de la UI.

### 5. Renderizado de la Pantalla

Finalmente, se llama a `TaskListScreen`, pasando el `viewModel` y los modificadores:

```
TaskListScreen(viewModel = viewModel, modifier = modifier)
```

Esta es la pantalla que realmente mostrará las tareas al usuario.





---

## Definición de la Clase Task

Esta es una **data class**, lo que significa que Kotlin generará automáticamente funciones útiles como `toString()`, `equals()`, `hashCode()`, y `copy()`.

```
data class Task(
    val id: Int? = null,
    val title: String,
    val description: String,
    val date: String
)
```

Cada tarea tiene los siguientes atributos:

1. **id: Int? = null** 
  - Es un identificador único para cada tarea.
  - Es nullable ( Int? ), lo que significa que puede ser null cuando se crea una nueva tarea (antes de que la API le asigne un ID).
2. **title: String** 
  - Representa el título de la tarea.
  - Es un String, por lo que no puede ser null.
3. **description: String** 
  - Contiene detalles adicionales sobre la tarea.
  - También es un String obligatorio.
4. **date: String** 
  - Guarda la fecha de la tarea en formato String.
  - Podría almacenarse en formatos como "2025-03-10" o "10/03/2025".

Esta clase es fundamental en la aplicación, ya que será utilizada por el repositorio, el ViewModel y la UI. Si quieres, ahora podemos analizar la TaskApi o TaskRepository para ver cómo se interactúa con este modelo en la API. 🚀

## Estructura de TaskApi

Esta interfaz define cinco operaciones principales sobre las tareas:

1. **Obtener todas las tareas ( GET /tasks )** 

```
@GET("tasks")
suspend fun getTasks(): List<Task>
```

- Realiza una petición GET al endpoint /tasks y devuelve una lista de Task.
- suspend indica que esta función se ejecuta dentro de una **corutina** para evitar bloqueos en la UI.
- Ejemplo de respuesta del backend:

```
[
  { "id": 1, "title": "Comprar café", "description": "Ir al supermercado",
    "date": "2025-03-10" },
  { "id": 2, "title": "Estudiar Kotlin", "description": "Repasar ViewModel y
    Compose", "date": "2025-03-11" }
]
```

## 2. Obtener una tarea por ID ( GET /tasks/{id} ) 🔍

```
@GET("tasks/{id}")
suspend fun getTask(@Path("id") id: Int): Task
```

- Devuelve una tarea específica según su `id`.
- `@Path("id")` indica que el valor del `id` se insertará dinámicamente en la URL ( `/tasks/1` ).
- Ejemplo de respuesta:

```
{ "id": 1, "title": "Comprar café", "description": "Ir al supermercado",
  "date": "2025-03-10" }
```

## 3. Crear una nueva tarea ( POST /tasks ) ➕

```
@POST("tasks")
suspend fun createTask(@Body task: Task): Task
```

- Envía un objeto `Task` al backend para ser guardado.
- `@Body` convierte el objeto Kotlin en JSON automáticamente.
- Ejemplo de solicitud enviada al servidor:

```
{ "title": "Ir al gimnasio", "description": "Entrenar 1 hora", "date":
  "2025-03-12" }
```

- Ejemplo de respuesta del backend (con el ID asignado):

```
{ "id": 3, "title": "Ir al gimnasio", "description": "Entrenar 1 hora",
  "date": "2025-03-12" }
```

## 4. Actualizar una tarea ( PUT /tasks/{id} ) ✎

```
@PUT("tasks/{id}")
suspend fun updateTask(@Path("id") id: Int, @Body task: Task): Task
```

- Modifica una tarea existente enviando el `id` en la URL y los nuevos datos en el cuerpo (`@Body`).
- Ejemplo de solicitud para actualizar la tarea con `id=1`:

```
{ "id": 1, "title": "Comprar café", "description": "Ir a la tienda de especialidad", "date": "2025-03-10" }
```

- Ejemplo de respuesta:

```
{ "id": 1, "title": "Comprar café", "description": "Ir a la tienda de especialidad", "date": "2025-03-10" }
```

## 5. Eliminar una tarea (DELETE /tasks/{id}) ❌

```
@DELETE("tasks/{id}")
suspend fun deleteTask(@Path("id") id: Int)
```

- Elimina la tarea con el `id` especificado.
- No devuelve respuesta (`Unit`).
- Uso en código:

---

Esta interfaz `TaskApi` define todas las operaciones necesarias para comunicarse con la API de tareas mediante Retrofit.

## Explicación de TaskListScreen

Esta función es la pantalla principal donde se muestran todas las tareas.

```
@Composable
fun TaskListScreen(viewModel: TaskViewModel, modifier: Modifier = Modifier) {
    val tasks by viewModel.tasks.collectAsState()
```

- **Recupera el estado de las tareas:**
  - `viewModel.tasks.collectAsState()` convierte el `Flow` de tareas en un estado reactivo dentro de Compose.
  - `by` simplifica el acceso a los valores del estado.
  - `tasks` contiene la lista de tareas recuperadas desde el `ViewModel`.

## Estructura de la Lista

```
Column(modifier = modifier.fillMaxSize()) {
    LazyColumn(
        contentPadding = PaddingValues(16.dp),
        verticalArrangement = Arrangement.spacedBy(8.dp)
    ) {
        items(tasks) { task ->
            TaskItem(task = task)
        }
    }
}
```

- **Column** 📦: Estructura principal que contiene la lista.
- **LazyColumn** 📖: Lista optimizada que carga elementos de manera eficiente a medida que se desplazan en pantalla.
  - `contentPadding = PaddingValues(16.dp)`: Agrega un margen alrededor de la lista.
  - `verticalArrangement = Arrangement.spacedBy(8.dp)`: Espaciado vertical de 8.dp entre elementos.
- **items(tasks) { task -> TaskItem(task) }**
  - Itera sobre la lista de tareas y para cada una llama a `TaskItem`.

## 🚀 Carga de Tareas

```
viewModel.loadTasks()
```

- **Carga las tareas** inmediatamente al mostrar la pantalla.
- `viewModel.loadTasks()` obtiene las tareas desde la API y actualiza el estado de `tasks`.

## 📝 Explicación de TaskItem

Cada tarea se representa dentro de una tarjeta ( `Card` ).

```
@Composable
fun TaskItem(task: Task) {
    Card(
        modifier = Modifier
            .fillMaxWidth()
            .padding(vertical = 4.dp),
        elevation = CardDefaults.cardElevation(defaultElevation = 4.dp)
    ) {
```

- **Card** 📌: Contenedor visual con sombra para destacar cada tarea.
- **modifier = Modifier.fillMaxWidth().padding(vertical = 4.dp)**

- Ocupar todo el ancho disponible ( `fillMaxWidth()` ).
- Agregar un espacio de `4.dp` arriba y abajo ( `padding(vertical = 4.dp)` ).
- **elevation = CardDefaults.cardElevation(defaultElevation = 4.dp)**
  - Agrega un efecto de sombra para resaltar la tarjeta.

## Contenido de la Tarjeta

```
Column(
    modifier = Modifier.padding(16.dp)
) {
    Text(
        text = task.title ?: "Nombre no disponible",
        style = MaterialTheme.typography.headlineSmall
    )
    Spacer(modifier = Modifier.height(8.dp))
    Text(
        text = task.description ?: "Descripción no disponible",
        style = MaterialTheme.typography.bodyMedium
    )
    Spacer(modifier = Modifier.height(8.dp))
    Text(
        text = task.date ?: "Fecha no disponible",
        style = MaterialTheme.typography.bodySmall
    )
}
```

- **Column** : Organiza los textos en vertical con un padding interno de `16.dp` .
- **Text (Título, Descripción, Fecha)**
  - Se usa `MaterialTheme.typography.headlineSmall` para el título.
  - Se deja espacio con `Spacer(modifier = Modifier.height(8.dp))` entre elementos.
  - Si un valor es `null` , se muestra un texto por defecto como "Nombre no disponible" .

## Ejemplo de Uso


Supongamos que `TaskListScreen` recibe un `ViewModel` que contiene estas tareas:

```
val exampleTasks = listOf(
    Task(id = 1, title = "Estudiar Kotlin", description = "Repasar ViewModel y Compose", date = "2025-03-11"),
    Task(id = 2, title = "Hacer ejercicio", description = "30 minutos de cardio", date = "2025-03-12")
)
```


Si `viewModel.tasks` contiene esta lista, la pantalla se vería así:



## Estudiar Kotlin


- Repasar ViewModel y Compose
-  2025-03-11

## Hacer ejercicio

- 30 minutos de cardio
  -  2025-03-12
- 

## Explicación de TaskRepository

```
class TaskRepository(private val api: TaskApi) {
```

- **Recibe una instancia de TaskApi** 
    - TaskApi es la interfaz que define las operaciones de red con Retrofit.
    - TaskRepository encapsula esta API para que otras partes de la aplicación puedan acceder a las tareas sin interactuar directamente con Retrofit.
    - El uso de un **repositorio** es una buena práctica en arquitectura, ya que permite separar la lógica de datos de la UI.
- 

## Funciones del Repositorio

Cada función delega la llamada a la API de Retrofit, pero proporciona un punto centralizado para manejar los datos.

### Obtener todas las tareas

```
suspend fun getTasks(): List<Task> = api.getTasks()
```

- **Devuelve una lista de tareas ( List<Task> )** recuperada desde el backend.
  - **Es suspend** , lo que significa que se ejecuta en una **corrutina** y no bloquea el hilo principal.
- 

### Obtener una tarea por ID

```
suspend fun getTask(id: Int): Task = api.getTask(id)
```

- Recibe un `id` y llama a `api.getTask(id)` .

- Retorna un objeto `Task` con la información de la tarea.

### Ejemplo de Uso:

Si llamamos a `getTask(3)`, podríamos obtener:

```
{
  "id": 3,
  "title": "Comprar leche",
  "description": "Recordar comprar leche en el supermercado",
  "date": "2025-03-12"
}
```

---

## + Crear una nueva tarea

```
suspend fun createTask(task: Task): Task = api.createTask(task)
```

- Envía un objeto `Task` al backend y recibe la tarea creada.
- Normalmente, el backend asigna un `id` único a la nueva tarea.

### Ejemplo de Uso:

```
val nuevaTarea = Task(title = "Leer un libro", description = "Leer 30 páginas al día", date = "2025-03-15")
val tareaCreada = repository.createTask(nuevaTarea)
```

- Supongamos que el backend devuelve esta respuesta:

```
{
  "id": 5,
  "title": "Leer un libro",
  "description": "Leer 30 páginas al día",
  "date": "2025-03-15"
}
```

- `tareaCreada` ahora contiene la misma tarea, pero con `id = 5`.

---

## ⇒ Actualizar una tarea

```
suspend fun updateTask(id: Int, task: Task): Task = api.updateTask(id, task)
```

- Recibe un `id` y un objeto `Task` con la información actualizada.

- Llama a `api.updateTask(id, task)`, que envía la actualización al backend.
- Retorna la versión actualizada de la tarea.

#### Ejemplo de Uso:

```
val tareaEditada = Task(title = "Leer un libro", description = "Leer 50 páginas al día", date = "2025-03-16")
val resultado = repository.updateTask(5, tareaEditada)
```

**Si la API responde correctamente, `resultado` ahora tiene la tarea con los cambios aplicados.**

#### Eliminar una tarea

```
suspend fun deleteTask(id: Int) = api.deleteTask(id)
```




- Recibe el `id` de la tarea y la elimina llamando a `api.deleteTask(id)`.
- No retorna nada porque la API normalmente solo confirma que la tarea fue eliminada.

#### Ejemplo de Uso:

```
repository.deleteTask(5)
```

- Si el backend responde con un `200 OK`, significa que la tarea fue eliminada con éxito.

### Resumen

-  **TaskRepository** actúa como puente entre la UI y la API.
-  **Separa la lógica de red de la lógica de la aplicación**, manteniendo el código más limpio.
-  **Uso de `suspend` permite que todas las funciones sean llamadas de forma asíncrona**, evitando bloqueos en la UI.

Este código es clave en la arquitectura, ya que evita que `TaskViewModel` interactúe directamente con Retrofit.

### Estructura de TaskViewModel

```
class TaskViewModel(private val repository: TaskRepository) : ViewModel()
```

- **Extiende `ViewModel`** , lo que significa que sobrevive a cambios de configuración (como rotación de pantalla).
  - **Recibe un `TaskRepository` como dependencia**, que se usará para interactuar con el backend.
- 

## Estados: `MutableStateFlow` y `StateFlow`

```
private val _tasks = MutableStateFlow<List<Task>>(emptyList())
val tasks: StateFlow<List<Task>> = _tasks
```

- `_tasks` es una **flujo mutable** (`MutableStateFlow`) que mantiene una lista de tareas.
- `tasks` es la **versión inmutable** (`StateFlow`) a la que accederá la UI.
- **Los `StateFlow` permiten que la UI se actualice automáticamente** cuando los datos cambian.

### ◆ Ejemplo en Compose:

En `TaskListScreen`, usamos `collectAsState()` para escuchar cambios en `tasks`:

```
val tasks by viewModel.tasks.collectAsState()
```

---

## Carga inicial de tareas ( `init` )

```
init {
    loadTasks()
}
```

- Al crear el `ViewModel`, se ejecuta `loadTasks()` automáticamente.
  - Esto significa que la pantalla siempre empieza con la lista de tareas actualizada.
- 

## Funciones para gestionar tareas

Cada función usa **corutinas** ( `viewModelScope.launch` ) para que las llamadas a la API sean asíncronas.

### Cargar todas las tareas

```
fun loadTasks() {
    viewModelScope.launch {
        _tasks.value = repository.getTasks()
    }
}
```

- Llama a `repository.getTasks()`, que obtiene la lista desde la API.
  - **Actualiza** `_tasks.value`, lo que automáticamente actualiza la UI.
- 

## Cargar una tarea específica

```
fun loadTask(id: Int) {
    viewModelScope.launch {
        _task.value = repository.getTask(id)
    }
}
```

- **Recibe un ID y obtiene la tarea correspondiente.**
- `_task` es otro `StateFlow<Task?>`, útil para **mostrar detalles de una tarea** en otra pantalla.

### ◆ Ejemplo de uso en una pantalla de detalles:

```
val task by viewModel.task.collectAsState()
Text(text = task?.title ?: "Cargando...")
```

- La UI se actualizará automáticamente cuando la tarea se cargue.
- 

## + Crear una nueva tarea

```
fun createTask(task: Task) {
    viewModelScope.launch {
        repository.createTask(task)
        loadTasks() // Recargar la lista
    }
}
```

- **Crea una nueva tarea** en el backend.
- Luego, llama a `loadTasks()` para **actualizar la lista en la UI**.

### ◆ Ejemplo en un formulario:

```
viewModel.createTask(Task(title = "Nueva tarea", description = "Descripción", date = "2025-03-15"))
```

---

## ✎ Actualizar una tarea

```
fun updateTask(id: Int, task: Task) {
    viewModelScope.launch {
        repository.updateTask(id, task)
        loadTasks() // Recargar la lista
    }
}
```

- Envía la actualización al backend y **recarga la lista**.
- Similar a `createTask`, esto asegura que la UI refleje los cambios.

### ◆ Ejemplo de actualización:

```
viewModel.updateTask(3, Task(title = "Tarea editada", description = "Nuevo contenido", date = "2025-03-16"))
```

---

## 🗑 Eliminar una tarea

```
fun deleteTask(id: Int) {
    viewModelScope.launch {
        repository.deleteTask(id)
        loadTasks() // Recargar la lista
    }
}
```

- **Elimina una tarea del backend.**
- Luego, **recarga la lista** para reflejar la eliminación en la UI.

### ◆ Ejemplo de eliminación desde un botón:

```
Button(onClick = { viewModel.deleteTask(3) }) {
    Text("Eliminar tarea")
}
```

---

## Factory para ViewModelProvider

```
class Factory(private val repository: TaskRepository) : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom(TaskViewModel::class.java)) {
            @Suppress("UNCHECKED_CAST")
            return TaskViewModel(repository) as T
        }
        throw IllegalArgumentException("Unknown ViewModel class")
    }
}
```

- Crea una instancia de TaskViewModel y le pasa el TaskRepository.
- Es necesaria para inicializar TaskViewModel correctamente en TaskListScreenWrapper.

### ◆ Ejemplo de uso en TaskListScreenWrapper :

```
val viewModel: TaskViewModel = viewModel(factory =
TaskViewModel.Factory(repository))
```

Así, el ViewModel se crea con el repositorio correctamente inyectado.

## Resumen

- ✓ TaskViewModel administra los datos de la UI y mantiene la lista de tareas sincronizada.
- ✓ Usa MutableStateFlow para que los cambios en los datos **actualicen automáticamente la UI**.
- ✓ Utiliza viewModelScope.launch para ejecutar llamadas a la API **sin bloquear el hilo principal**.
- ✓ Incluye una Factory para la creación correcta del ViewModel.

Con este ViewModel, la aplicación es **reactiva**, eficiente y organizada. ¡Tu app de gestión de tareas está bien estructurada! 🚀