Side-Effect Handler APIs

다이스

Side-Effect Hanlder API란?

Composable의 ReComposition 범위 외부에서 작업을 처리할 수 있도록 여러 API를 제공

✅ 안드로이드 프레임워크와 상호 작용

✅ 컴포지션 이벤트 관리

✅ 상태 변경 기반 사이드 이펙트 처리

LaunchedEffect / DisposableEffect / SideEffect

LaunchedEffect

Key 값이 변경될 때마다 이하 block 영역을 비동기로 실행

Effects.kt

```kotlin
@Composable
@NonRestartableComposable
@Suppress( ...names: "ArrayReturn")
@OptIn(InternalComposeApi::class)
fun LaunchedEffect(
    vararg keys: Any?,
    block: suspend CoroutineScope.() -> Unit
) {
    val applyContext = currentComposer.applyCoroutineContext
    remember(*keys) { LaunchedEffectImpl(applyContext, block) }
}
```

LaunchedEffect

Effects.kt

```kotlin
internal class LaunchedEffectImpl(
    parentCoroutineContext: CoroutineContext,
    private val task: suspend CoroutineScope.() -> Unit
) : RememberObserver {
    private val scope = CoroutineScope(parentCoroutineContext)
    private var job: Job? = null

    override fun onRemembered() {
        // This should never happen but is left here for safety
        job?.cancel( message: "Old job was still running!")
        job = scope.launch(block = task)
    }

    override fun onForgotten() {
        job?.cancel(LeftCompositionCancellationException())
        job = null
    }

    override fun onAbandoned() {
        job?.cancel(LeftCompositionCancellationException())
        job = null
    }
}
```

onRemembered :
Composition에 들어갈 때,
새로운 코루틴을 열어 task 실행

onForgotten :
컴포저블이 Composition에서 빠질 때,
실행 중이던 코루틴 취소

onAbandoned :
컴포저블이 재사용되지 않고 버려질 때,
실행 중이던 코루틴 취소

LaunchedEffect

화면 진입 시 데이터 로딩

리스트 아이템 클릭 후 네트워크 요청

애니메이션 실행

...

MainActivity.kt

```kotlin
@Composable
fun MyComposable() {
    val isLoading = remember { mutableStateOf( value: false) }
    val data = remember { mutableStateOf(listOf<String>()) }

    LaunchedEffect(isLoading.value) {
        if (isLoading.value) {
            val newData = fetchData()
            data.value = newData
            isLoading.value = false
        }
    }

    Column {
        Button(onClick = { isLoading.value = true }) {
            Text( text: "Fetch Data")
        }
        if (isLoading.value) {
            CircularProgressIndicator()
        } else {
            LazyColumn {
                items(data.value.size) { index ->
                    Text(text = data.value[index])
                }
            }
        }
    }
}
```

DisposableEffect

부모 컴포저블이 처음 렌더링 될 때 side effect를 실행하고,
컴포저블이 UI 계층에서 제거될 시에 effect를 폐기

Effects.kt

```kotlin
@Composable
@NonRestartableComposable
@Suppress( ...names: "ArrayReturn")
fun DisposableEffect(
    vararg keys: Any?,
    effect: DisposableEffectScope.() -> DisposableEffectResult
) {
    remember(*keys) { DisposableEffectImpl(effect) }
}
```

DisposableEffect

Effects.kt

```
private class DisposableEffectImpl(
    private val effect: DisposableEffectScope.() -> DisposableEffectResult
) : RememberObserver {
    private var onDispose: DisposableEffectResult? = null

    override fun onRemembered() {
        onDispose = InternalDisposableEffectScope.effect()
    }

    override fun onForgotten() {
        onDispose?.dispose()
        onDispose = null
    }

    override fun onAbandoned() {
        // Nothing to do as [onRemembered] was not called.
    }
}
```

DisposableEffect

Effects.kt

```
class DisposableEffectScope {

    Provide onDisposeEffect to the DisposableEffect to run when it leaves
    key changes.

    inline fun onDispose(
        crossinline onDisposeEffect: () -> Unit
    ): DisposableEffectResult = object : DisposableEffectResult {
        override fun dispose() {
            onDisposeEffect()
        }
    }
}
```

DisposableEffect

이벤트 리스너를 붙이거나 애니메이션을 실행시키는 것처럼 컴포저블 함수가 UI 계층에서 제거될 때 사용되지 않아야 하는 로직들을 실행시켜야 할 때 유용

MainActivity.kt

```kotlin
@Composable
fun MyComposable(lifecycleOwner: LifecycleOwner = LocalLifecycleOwner.current) {
    DisposableEffect(lifecycleOwner) {
        val observer = LifecycleEventObserver { _, event -> /* ... */ }
        lifecycleOwner.lifecycle.addObserver(observer)
        onDispose { lifecycleOwner.lifecycle.removeObserver(observer) }
    }
}
```

SideEffect

현재 컴포지션이 성공적으로 완료되고 변경 사항이 적용된 직후 실행됨
컴포저블의 상태나 Props에 의존하지 않는 연산을 수행할 때 유용

Effects.kt

```kt
@Composable
@NonRestartableComposable
@ExplicitGroupsComposable
@OptIn(InternalComposeApi::class)
fun SideEffect(
    effect: () -> Unit
) {
    currentComposer.recordSideEffect(effect)
}
```

SideEffect

왜 필요한가?

Compose가 snapshot 기반 상태 관리를 하는데,

SideEffect는 snapshot으로 관리되지 않는 외부 객체와의 동기화에 필요

만약 컴포지션이 실패했는데 외부 객체를 먼저 변경하면 불일치 상태 발생 위험

→ 이를 방지

SideEffect

MainActivity.kt

```kotlin
@Composable
fun Counter(modifier: Modifier = Modifier) {
    var count by remember { mutableIntStateOf( value: 0) }

    SideEffect {
        Log.d( tag: "Counter",  msg: "Count is $count")
    }

    Column(modifier = modifier) {
        Button(onClick = { count++ }) {
            Text( text: "Increase Count $count")
        }
    }
}
```

```
D  Count is 0
```

여러 번 버튼을 클릭해도
로그 출력은 한 번 뿐

이유는?

SideEffect

SideEffect는 현재 컴포저블이 Recompose될 때만 실행되며,
중첩된 하위 컴포저블의 Recompose에는 영향을 받지 않음

즉, 내부 컴포저블이 Recompose되더라도
외부 컴포저블에 선언된 SideEffect는 실행되지 않음

Side-Effect Handler APIs 요약

API	주요 목적	사용 시점
LaunchedEffect	코루틴 실행, suspend 함수 호출	Key 변경, 컴포지션 진입 시
DisposableEffect	리소스 등록/해제, 정리	컴포지션 진입/해제 시
SideEffect	외부 시스템과 상태 동기화	매 recomposition 직후