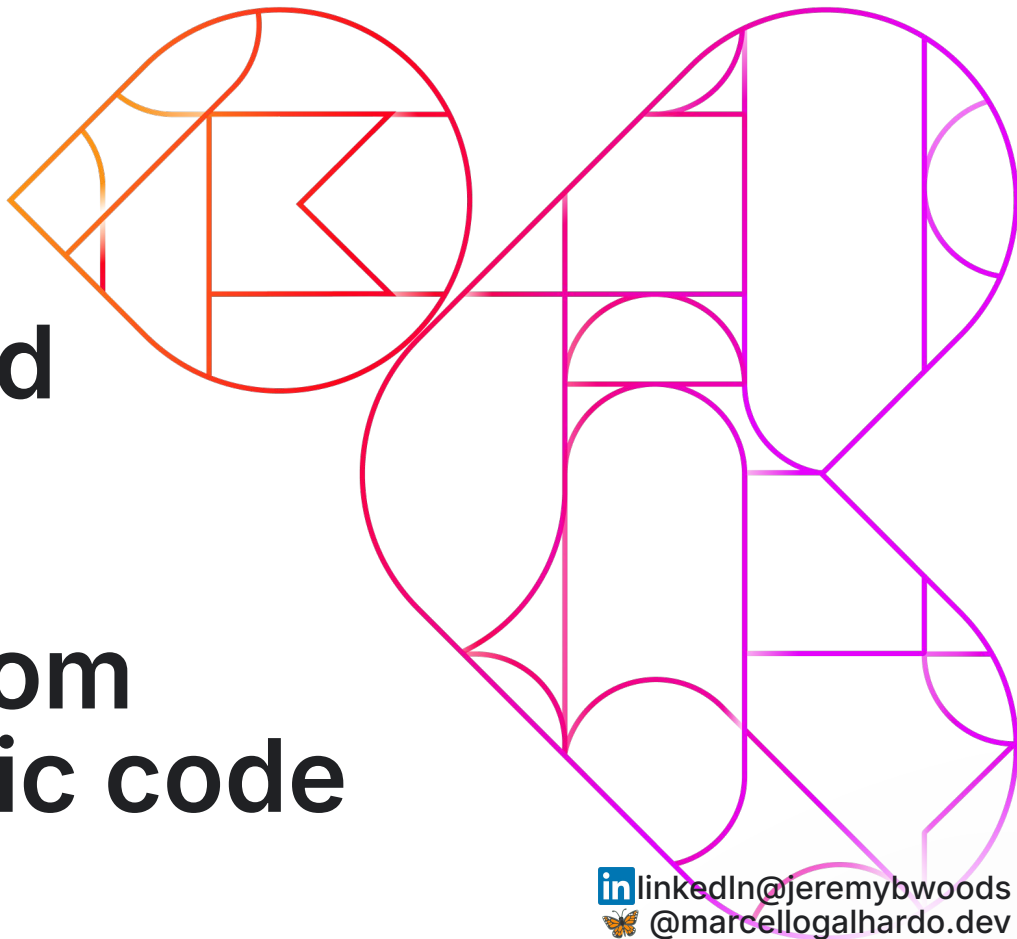


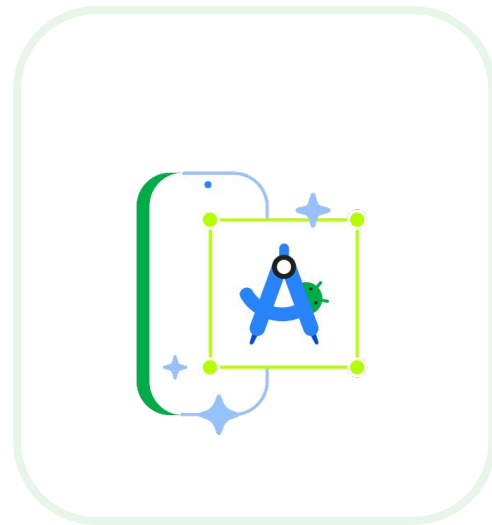
Lessons learned decoupling Architecture Components from platform specific code

Jeremy Woods, Marcello Galhardo

 [linkedin@jeremybwoods](#)
 [@marcellogalhardo.dev](#)

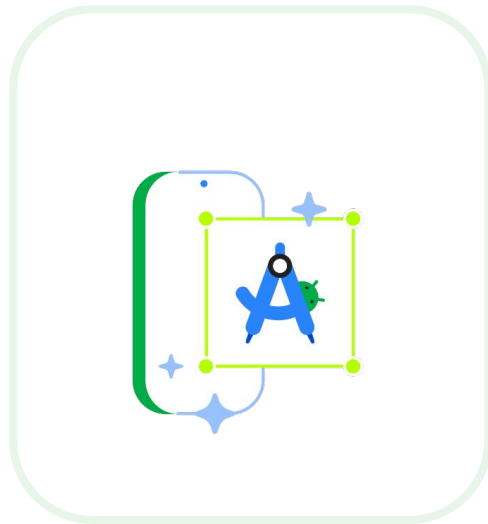


Architecture Components Team



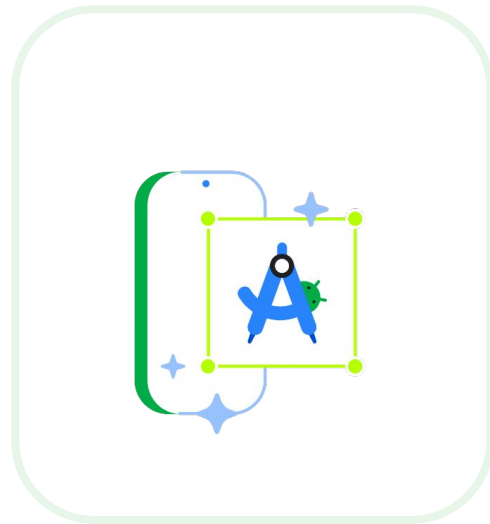
Architecture Components Team

- 6 people across 3 countries.



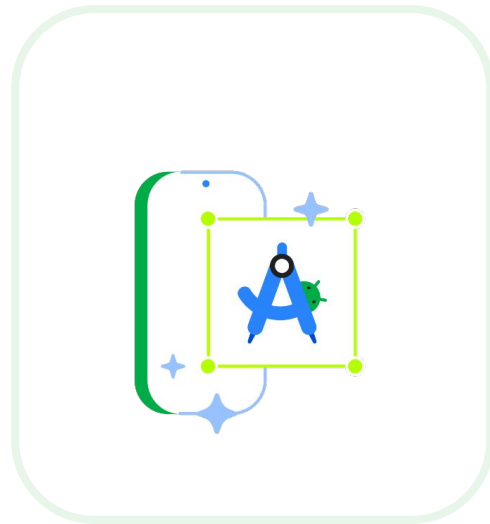
Architecture Components Team

- 6 people across 3 countries.
- 12 different libraries
 - Activity, DataStore, Fragment, Hilt, Lifecycle, Navigation, Paging, Room, SavedState, ViewModel, SQLite, XProcessing



Architecture Components Team

- 6 people across 3 countries.
- 14 different libraries
 - Activity, DataStore, Fragment, Hilt, Lifecycle, Navigation, Paging, Room, SavedState, ViewModel, SQLite, XProcessing
 - Navigation3, NavigationEvent



Today's agenda

01

02

03

04

Today's agenda

01 Motivation: why KMP?

02

03

04

Today's agenda

01 Motivation: why KMP?

02 Migrating an existing library

03

04

Today's agenda

01 Motivation: why KMP?

02 Migrating an existing library

03 Creating a new library

04

Today's agenda

01 Motivation: why KMP?

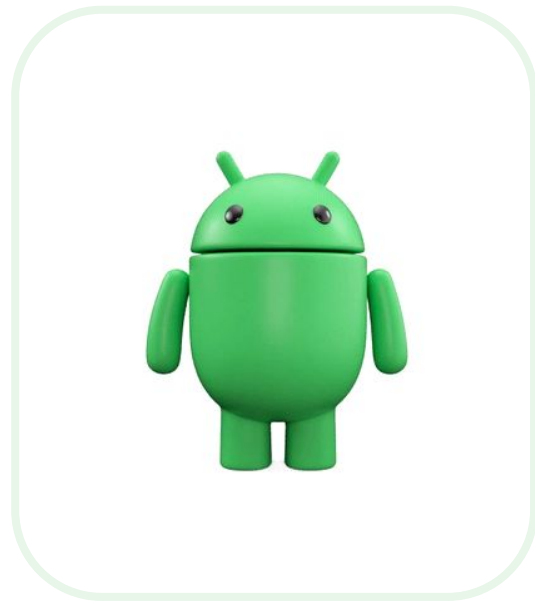
02 Migrating an existing library

03 Creating a new library

04 Review

01

Motivation: why **KMP**?

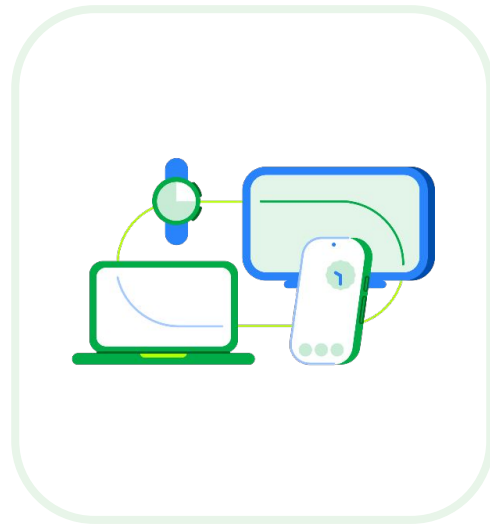


Why KMP?



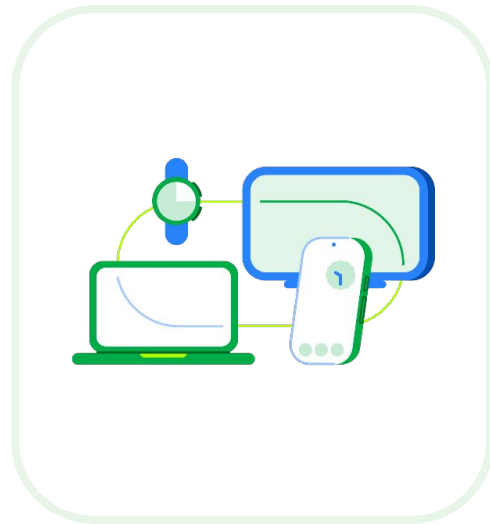
Why KMP?

- Android is Kotlin-first (since 2019).



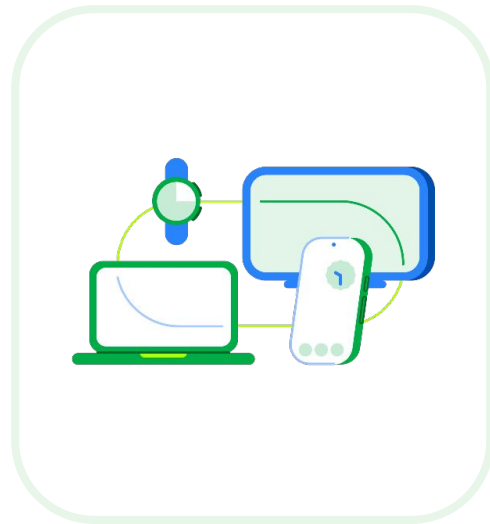
Why KMP?

- Android is Kotlin-first (since 2019).
- Code sharing across native platforms.



Why KMP?

- Android is Kotlin-first (since 2019).
- Code sharing across native platforms.
- Help building the ecosystem.



AndroidX Arch KMP Libraries

DataStore

Lifecycle

Navigation3

NavigationEvent

Paging

Room

SavedState

SQLite

ViewModel

AndroidX Arch KMP Libraries

DataStore

Lifecycle

Navigation3

NavigationEvent

Paging

Room

SavedState

SQLite

ViewModel

AndroidX Arch KMP Libraries

DataStore

Lifecycle

Navigation3

NavigationEvent

Paging

Room

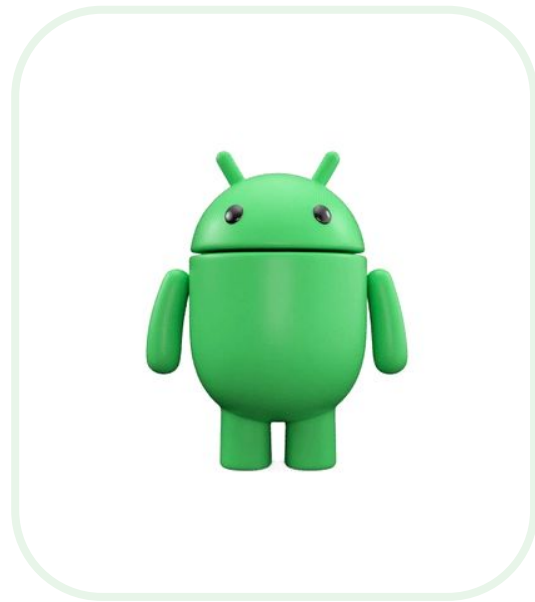
SavedState

SQLite

ViewModel

02

Migrating an **existing** library

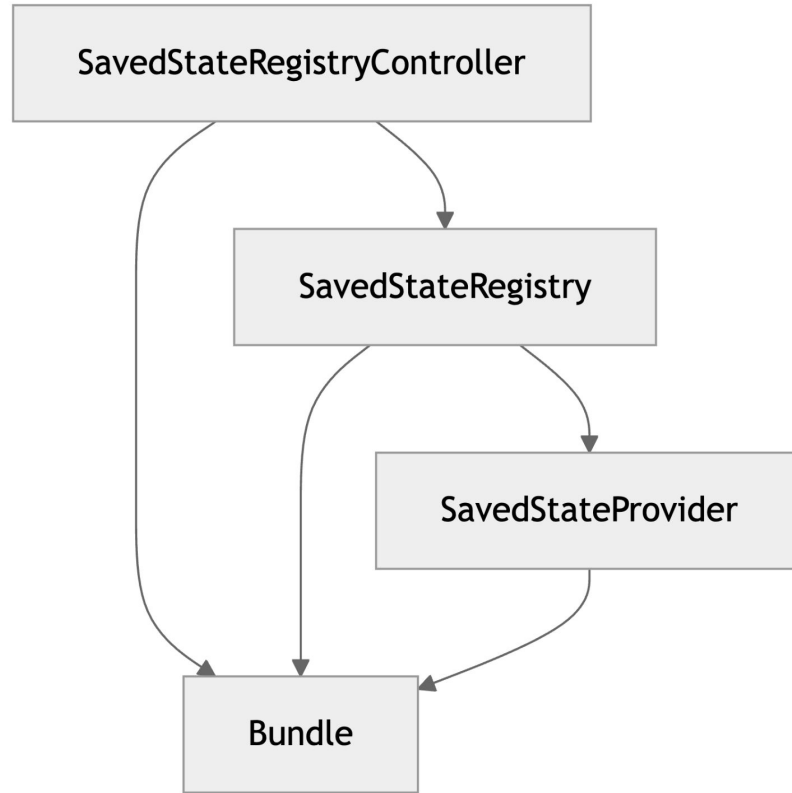


AndroidX SavedState

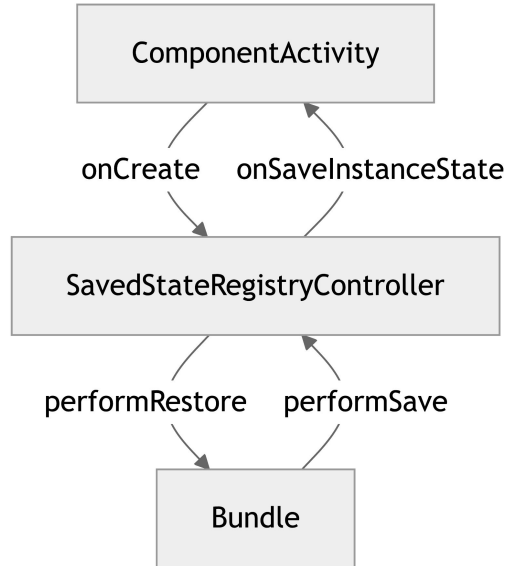
- Used to save and restore state in a process death.
- First alpha released in December 2018, written in Java.
- KMP-compatible release in May 2025 (1.3.0).



Core Components



Core Components



How these APIs look?

```
public class SavedStateRegistryController {  
  
    public val savedStateRegistry: SavedStateRegistry  
  
    public fun performRestore(savedState: Bundle?)  
    public fun performSave(outBundle: Bundle)  
  
    // {...}  
}
```

```
public class SavedStateRegistryController {  
  
    public val savedStateRegistry: SavedStateRegistry  
  
    public fun performRestore(savedState: Bundle?)  
    public fun performSave(outBundle: Bundle)  
  
    // {...}  
}
```

```
public class SavedStateRegistryController {  
  
    public val savedStateRegistry: SavedStateRegistry  
  
    public fun performRestore(savedState: Bundle?)  
    public fun performSave(outBundle: Bundle)  
  
    // {...}  
}
```

```
public class SavedStateRegistry {  
  
    public fun consumeRestoredStateForKey(key: String): Bundle?  
    public fun registerSavedStateProvider(key: String, provider: SavedStateProvider)  
  
    public fun interface SavedStateProvider {  
        public fun saveState(): Bundle  
    }  
    // {...}  
}
```

```
public class SavedStateRegistry {  
  
    public fun consumeRestoredStateForKey(key: String): Bundle?  
    public fun registerSavedStateProvider(key: String, provider: SavedStateProvider)  
  
    public fun interface SavedStateProvider {  
        public fun saveState(): Bundle  
    }  
    // {...}  
}
```

```
public class SavedStateRegistry {  
  
    public fun consumeRestoredStateForKey(key: String): Bundle?  
    public fun registerSavedStateProvider(key: String, provider: SavedStateProvider)  
  
    public fun interface SavedStateProvider {  
        public fun saveState(): Bundle  
    }  
    // {...}  
}
```

```
class ComponentActivity : Activity() {  
  
    val controller = SavedStateRegistryController.create(owner = this)  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        controller.performRestore(savedInstanceState)  
        // {...}  
    }  
  
    override fun onSaveInstanceState(outState: Bundle) {  
        // {...}  
        controller.performSave(outState)  
    }  
}
```

```
class ComponentActivity : Activity() {  
  
    val controller = SavedStateRegistryController.create(owner = this)  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        controller.performRestore(savedInstanceState)  
        // {...}  
    }  
  
    override fun onSaveInstanceState(outState: Bundle) {  
        // {...}  
        controller.performSave(outState)  
    }  
}
```

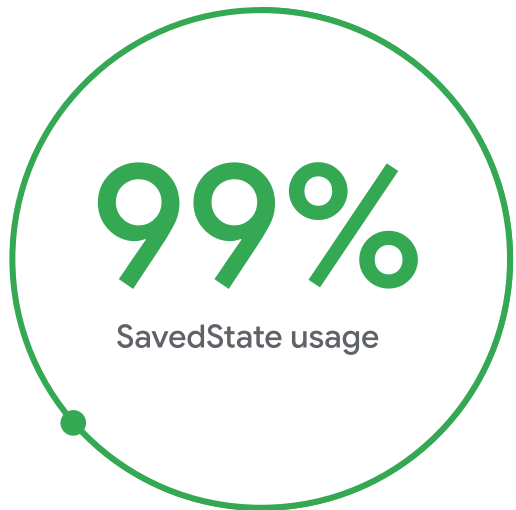
```
class ComponentActivity : Activity() {  
  
    val controller = SavedStateRegistryController.create(owner = this)  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        controller.performRestore(savedInstanceState)  
        // {...}  
    }  
  
    override fun onSaveInstanceState(outState: Bundle) {  
        // {...}  
        controller.performSave(outState)  
    }  
}
```

```
// Or a more high-level API.
```

```
class MyViewModel(  
    private val handle: SavedStateHandle,  
) : ViewModel() {  
  
    val name = handle.getMutableStateFlow("key") { "John Doe" }  
}
```

Why?

Top 50k apps using SavedState





```
public class SavedStateRegistryController {  
    // {...}  
    public fun performRestore(savedState: Bundle?)  
    public fun performSave(outBundle: Bundle)  
    // {...}  
}
```

```
public class SavedStateRegistryController {  
    // {...}  
    public fun performRestore(savedState: Bundle?)  
    public fun performSave(outBundle: Bundle)  
    // {...}  
}
```


Android Framework Team
!= AndroidX Team

**“Extension is shadowed by
a member”**

Zero-cost abstraction

```
public class SavedStateRegistryController {  
    // {...}  
    public fun performRestore(savedState: Bundle?)  
    public fun performSave(outBundle: Bundle)  
    // {...}  
}
```

```
public class SavedStateRegistryController {  
    // {...}  
    public fun performRestore(savedState: SavedState?)  
    public fun performSave(outBundle: SavedState)  
    // {...}  
}
```

Expect Actual

```
// commonMain
```

```
public expect class SavedState
```

```
// androidMain
```

```
public actual typealias SavedState = android.os.Bundle
```

```
// nonAndroidMain
```

```
public actual class SavedState(map: MutableMap<String, Any?>)
```

```
// commonMain
public expect class SavedState

// androidMain
public actual typealias SavedState = android.os.Bundle

// nonAndroidMain
public actual class SavedState(map: MutableMap<String, Any?>)
```

```
// commonMain
public expect class SavedState

// androidMain
public actual typealias SavedState = android.os.Bundle

// nonAndroidMain
public actual class SavedState(map: MutableMap<String, Any?>)
```

```
public expect inline fun savedState(block: SavedStateWriter.() -> Unit): SavedState
```

```
@JvmInline
```

```
public expect value class SavedStateReader(source: SavedState) // {...}
```

```
public inline fun <T> SavedState.read(block: SavedStateReader.() -> T): T // {...}
```

```
@JvmInline
```

```
public expect value class SavedStateWriter(source: SavedState) // {...}
```

```
public inline fun <T> SavedState.write(block: SavedStateWriter.() -> T): T // {...}
```

```
public expect inline fun savedState(block: SavedStateWriter.() -> Unit): SavedState
```

```
@JvmInline
```

```
public expect value class SavedStateReader(source: SavedState) // {...}
```

```
public inline fun <T> SavedState.read(block: SavedStateReader.() -> T): T // {...}
```

```
@JvmInline
```

```
public expect value class SavedStateWriter(source: SavedState) // {...}
```

```
public inline fun <T> SavedState.write(block: SavedStateWriter.() -> T): T // {...}
```

```
public expect inline fun savedState(block: SavedStateWriter.() -> Unit): SavedState
```

```
@JvmInline
```

```
public expect value class SavedStateReader(source: SavedState) // {...}
```

```
public inline fun <T> SavedState.read(block: SavedStateReader.() -> T): T // {...}
```

```
@JvmInline
```

```
public expect value class SavedStateWriter(source: SavedState) // {...}
```

```
public inline fun <T> SavedState.write(block: SavedStateWriter.() -> T): T // {...}
```

```
public expect inline fun savedState(block: SavedStateWriter.() -> Unit): SavedState
```

```
@JvmInline
```

```
public expect value class SavedStateReader(source: SavedState) // {...}
```

```
public inline fun <T> SavedState.read(block: SavedStateReader.() -> T): T // {...}
```

```
@JvmInline
```

```
public expect value class SavedStateWriter(source: SavedState) // {...}
```

```
public inline fun <T> SavedState.write(block: SavedStateWriter.() -> T): T // {...}
```

```
public expect inline fun savedState(block: SavedStateWriter.() -> Unit): SavedState
```

```
@JvmInline
```

```
public expect value class SavedStateReader(source: SavedState) // {...}
```

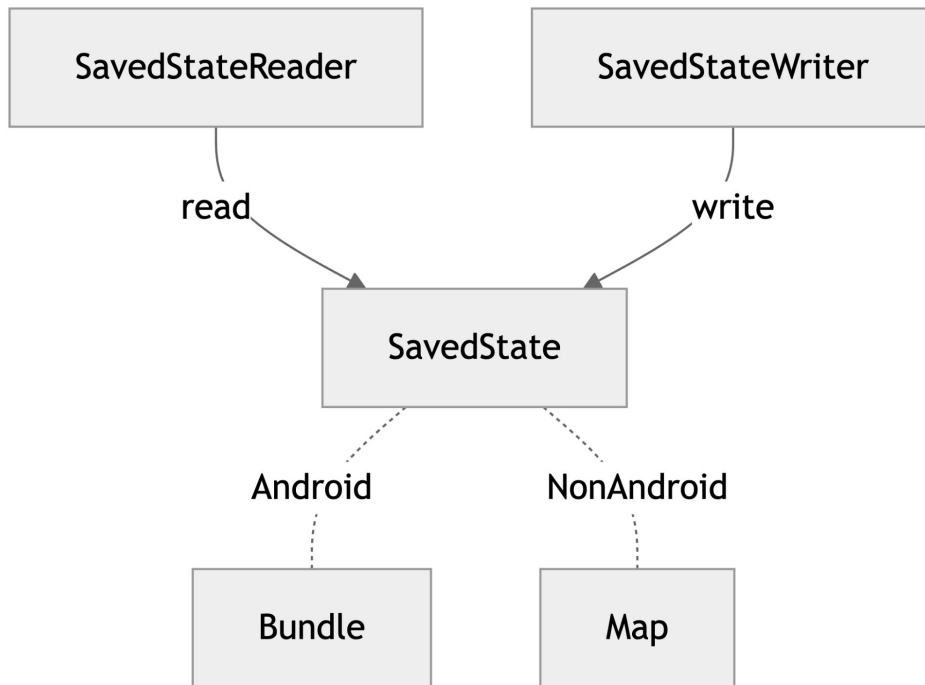
```
public inline fun <T> SavedState.read(block: SavedStateReader.() -> T): T // {...}
```

```
@JvmInline
```

```
public expect value class SavedStateWriter(source: SavedState) // {...}
```

```
public inline fun <T> SavedState.write(block: SavedStateWriter.() -> T): T // {...}
```

SavedState, Readers and Writers



```
data class Person(val name: String, val age: Int)
```

```
val person = Person("John Doe", 21)
```

```
val state = savedState { this: SavedStateWriter ->
    putString("name", person.name)
    putInt("age", person.age)
}
```

```
val name = state.read { this: SavedStateReader -> getString("name") }
```

```
data class Person(val name: String, val age: Int)
```

```
val person = Person("John Doe", 21)
```

```
val state = savedState { this: SavedStateWriter ->
    putString("name", person.name)
    putInt("age", person.age)
}
```

```
val name = state.read { this: SavedStateReader -> getString("name") }
```

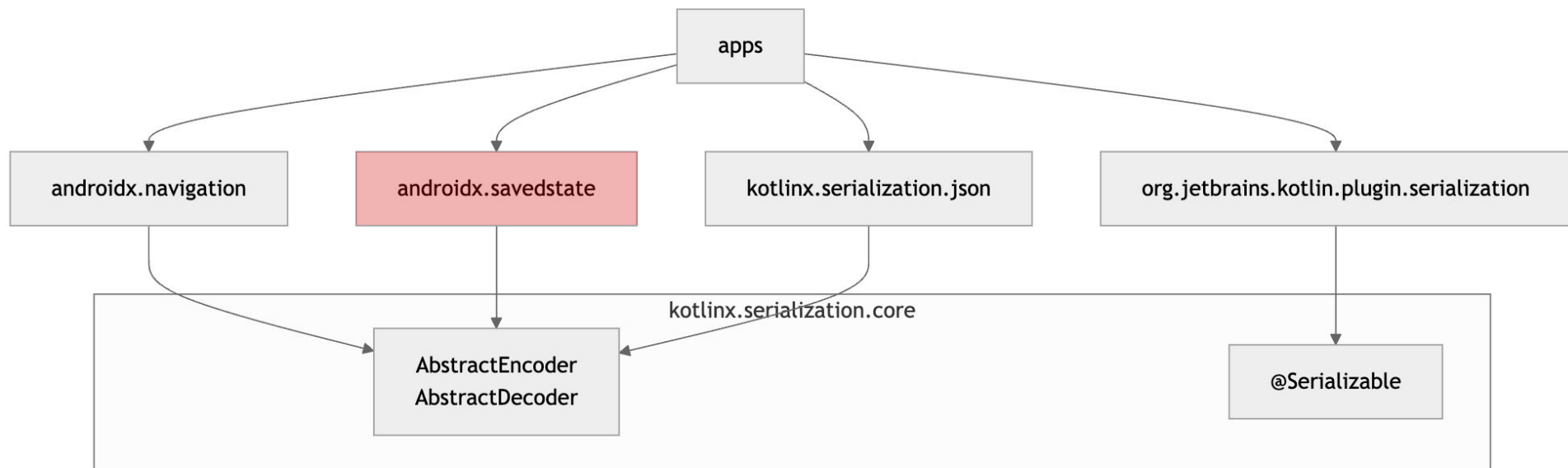
```
data class Person(val name: String, val age: Int)
```

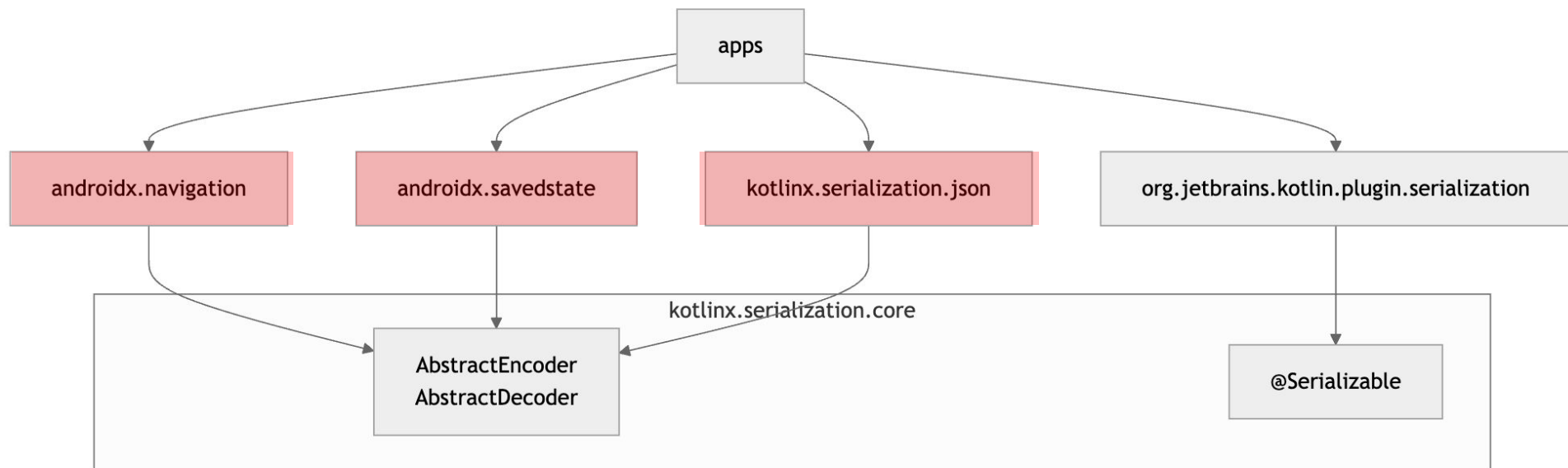
```
val person = Person("John Doe", 21)
```

```
val state = savedState { this: SavedStateWriter ->  
    putString("name", person.name)  
    putInt("age", person.age)  
}
```

```
val name = state.read { this: SavedStateReader -> getString("name") }
```

KotlinX Serialization





```
data class Person(val name: String, val age: Int)
```

```
@Serializable
```

```
data class Person(val name: String, val age: Int)
```

```
@Serializable
data class Person(val name: String, val age: Int)

fun main() {
    val person = Person("John Doe", 21)
    val encoded: SavedState = encodeToSavedState(person)
    val decoded: Person = decodeFromSavedState(encoded)
}
```

```
@Serializable
data class Person(val name: String, val age: Int)

fun main() {
    val person = Person("John Doe", 21)
    val encoded: SavedState = encodeToSavedState(person)
    val decoded: Person = decodeFromSavedState(encoded)
}
```

```
@Serializable
data class Person(val name: String, val age: Int)

fun main() {
    val person = Person("John Doe", 21)
    val encoded: SavedState = encodeToSavedState(person)
    val decoded: Person = decodeFromSavedState(encoded)
}
```

```
@Serializable
data class Person(val name: String, val age: Int)

fun main() {
    val person = Person("John Doe", 21)
    val encoded: SavedState = encodeToSavedState(person)
    val decoded: Person = decodeFromSavedState(encoded)
}
```

```
Bundle = [  
    "name": String = "John Doe",  
    "age" : Int    = 21,  
]
```

```
@Serializable
data class Person(val name: String, val age: Int)

fun main() {
    val configuration = SavedStateConfiguration {
        serializersModule = SerializersModule {...}
        encodeDefaults = true
        classDiscriminatorMode = ClassDiscriminatorMode.POLYMORPHIC
    }
    val person = Person("John Doe", 21)
    val encoded: SavedState = encodeToSavedState(configuration, person)
    val decoded: Person = decodeFromSavedState(configuration encoded)
}
```

```
@Serializable
```

```
data class Person(val name: String, val age: Int)
```

```
class MyViewModel(handle: SavedStateHandle) : ViewModel() {  
    val person by handle.saved { Person("John Doe", 21) }  
}
```

```
@Serializable  
data class Person(val name: String, val age: Int)  
  
class MyComponent : SavedStateRegistryOwner {  
    val person by saved { Person("John Doe", 21) }  
}
```

```
@Serializable  
data class Person(val name: String, val age: Int)  
  
class MyActivity : ComponentActivity() {  
    val person by saved { Person("John Doe", 21) }  
}
```

```
@Serializable
data class Person(val name: String, val age: Int)

class MyFragment : Fragment() {
    val person by saved { Person("John Doe", 21) }
}
```

```
@Serializable
```

```
data class Person(val name: String, val age: Int)
```

```
val navController = rememberNavController()
```

```
NavHost(navController, startDestination = Profile) {
```

```
    composable<Profile> { entry: NavBackStackEntry ->
```

```
        val selectedUser by remember { entry.saved { Person("John Doe", 21) } }
```

```
        Profile(navController)
```

```
    }
```

```
}
```

```
@Serializable
```

```
data class Person(val name: String, val age: Int)
```

```
val backStack = rememberNavBackStack()
```

```
NavDisplay(backStack, entryProvider = entryProvider {
```

```
    entry<Profile> {
```

```
        val owner = LocalSavedStateRegistryOwner.current.savedStateRegistry
```

```
        val selectedUser by remember { owner.saved { Person("John Doe", 21) } }
```

```
        Profile(navController)
```

```
    }
```

```
)
```

KotlinX Serialization + Compose

```
@Serializable
```

```
data class Person(val name: String, val age: Int)
```

```
@Composable
```

```
fun MyComposable() {
```

```
    val saveable = rememberSaveable(serializer = serializer<Person>()) {
```

```
        Person("John Doe", 21)
```

```
    }
```

```
}
```

```
@Serializable
```

```
data class Person(val name: String, val age: Int)
```

```
@Composable
```

```
fun MyComposable() {
```

```
    val saveable = rememberSaveable(serializer = serializer<Person>()) {
```

```
        Person("John Doe", 21)
```

```
    }
```

```
}
```

```
@Serializable
data class Person(val name: String, val age: Int)

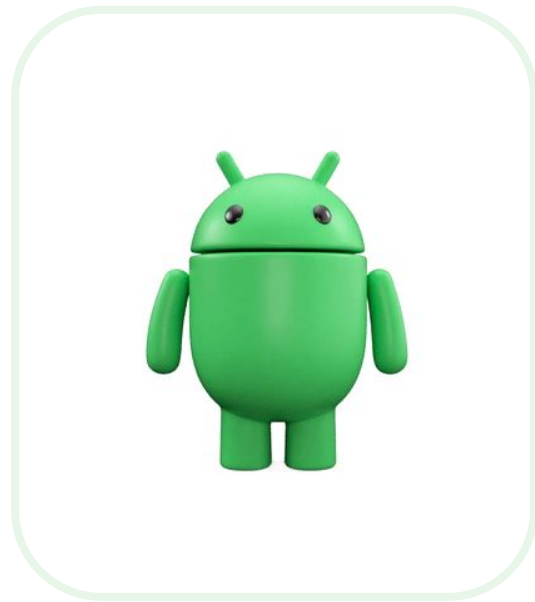
val backStack = rememberNavBackStack()
NavDisplay(backStack, entryProvider = entryProvider {
    entry<Profile> {
        val owner = LocalSavedStateRegistryOwner.current
        val registry = owner.savedStateRegistry
        val person by remember { registry.saved { Person("John Doe", 21) } }
        Profile(person)
    }
})
```

```
@Serializable
data class Person(val name: String, val age: Int)

val backStack = rememberNavBackStack()
NavDisplay(backStack, entryProvider = entryProvider {
    entry<Profile> {
        val person by rememberSaveable(serializer = serializer<Person>()) {
            Person("John Doe", 21)
        }
        Profile(person)
    }
})
```

03

Creating a new library for **KMP**



What is Navigation Event?



What is Navigation Event?

- System Back
 - Predictive Back



What is Navigation Event?

- System Back
 - Predictive Back
- Keyboard Events

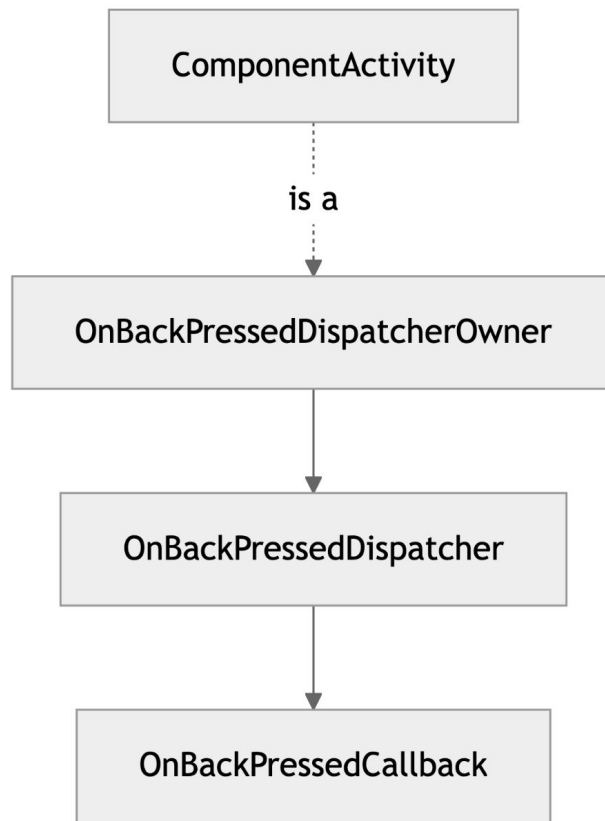


What is Navigation Event?

- System Back
 - Predictive Back
- Keyboard Events
- Forward Navigation

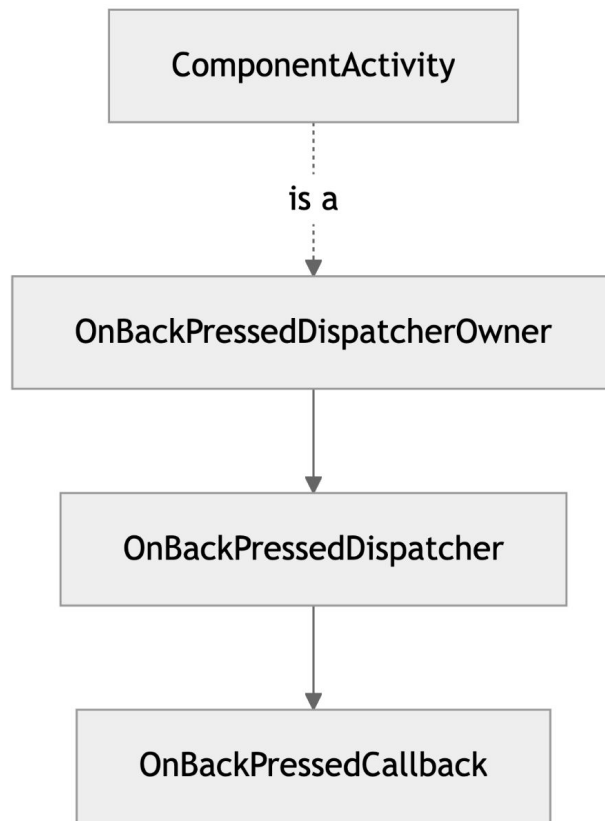


OnBackPressedDispatcher



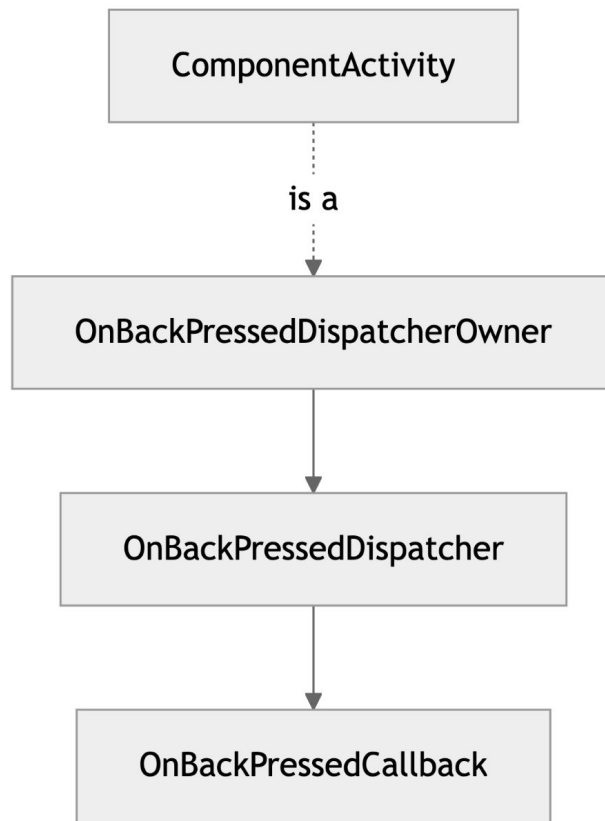
OnBackPressedDispatcher

- Added in the AndroidX Activity 1.0.0 release



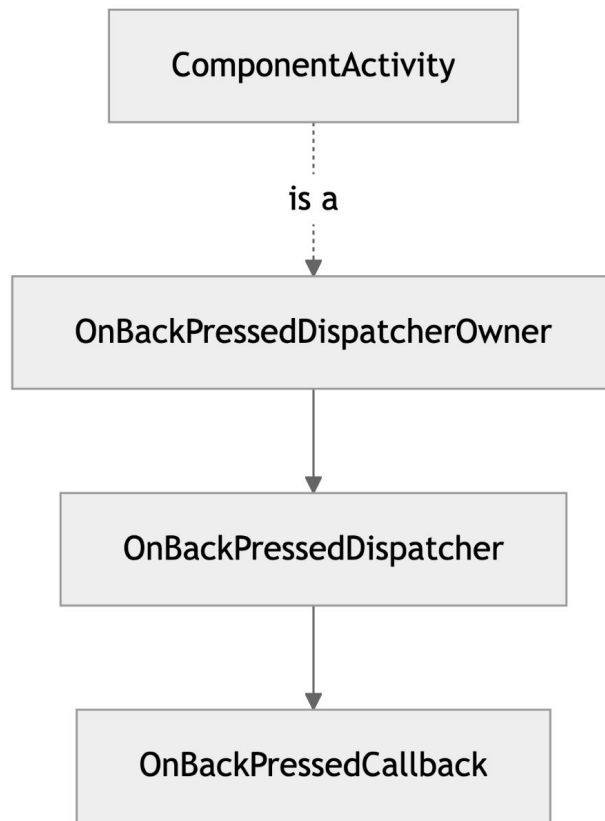
OnBackPressedDispatcher

- Added in the AndroidX Activity 1.0.0 release
- Available from ComponentActivity



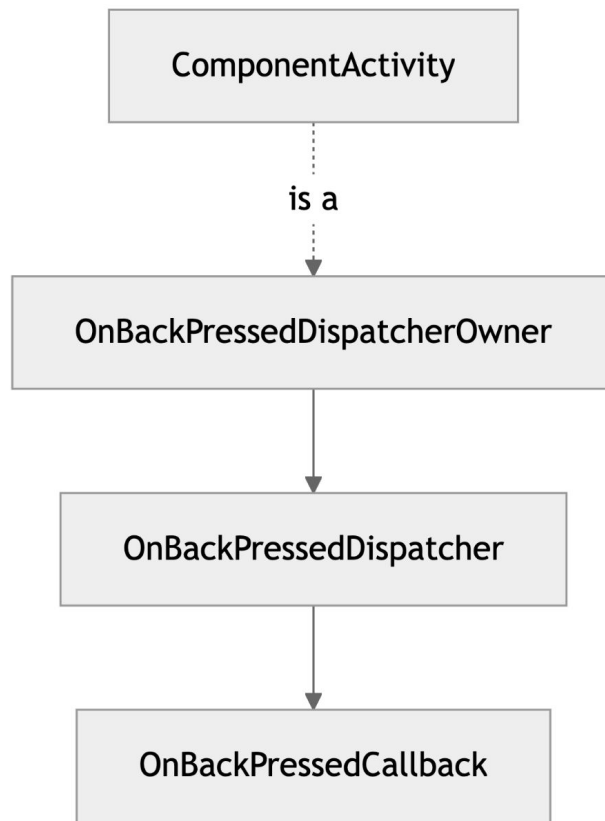
OnBackPressedDispatcher

- Added in the AndroidX Activity 1.0.0 release
- Available from ComponentActivity
- Register OnBackPressedCallback to listen for events



OnBackPressedDispatcher

- Added in the AndroidX Activity 1.0.0 release
- Available from ComponentActivity
- Register OnBackPressedCallback to listen for events
- Predictive/BackHandler in Compose



OnBackPressedDispatcher

```
class MyComponent(activity: ComponentActivity) {  
    val dispatcher = activity.onBackPressedDispatcher  
    init {  
        dispatcher.addCallback(object : OnBackPressedCallback(true) {  
            override fun handleOnBackPressedStarted(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressedProgressed(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressed() {...}  
            override fun handleOnBackPressedCancelled() {...}  
        })  
    }  
}
```

OnBackPressedDispatcher

```
class MyComponent(activity: ComponentActivity) {  
    val dispatcher = activity.onBackPressedDispatcher  
    init {  
        dispatcher.addCallback(object : OnBackPressedCallback(true) {  
            override fun handleOnBackPressedStarted(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressedProgressed(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressed() {...}  
            override fun handleOnBackPressedCancelled() {...}  
        })  
    }  
}
```

OnBackPressedDispatcher

```
class MyComponent(activity: ComponentActivity) {  
    val dispatcher = activity.onBackPressedDispatcher  
    init {  
        dispatcher.addCallback(object : OnBackPressedCallback(true) {  
            override fun handleOnBackPressedStarted(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressedProgressed(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressed() {...}  
            override fun handleOnBackPressedCancelled() {...}  
        })  
    }  
}
```

OnBackPressedDispatcher

```
class MyComponent(activity: ComponentActivity) {  
    val dispatcher = activity.onBackPressedDispatcher  
    init {  
        dispatcher.addCallback(object : OnBackPressedCallback(true) {  
            override fun handleOnBackPressedStarted(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressedProgressed(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressed() {...}  
            override fun handleOnBackPressedCancelled() {...}  
        })  
    }  
}
```

OnBackPressedDispatcher

```
class MyComponent(activity: ComponentActivity) {  
    val dispatcher = activity.onBackPressedDispatcher  
    init {  
        dispatcher.addCallback(object : OnBackPressedCallback(true) {  
            override fun handleOnBackStarted(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressedProgressed(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressed() {...}  
            override fun handleOnBackPressedCancelled() {...}  
        })  
    }  
}
```

OnBackPressedDispatcher

```
class MyComponent(activity: ComponentActivity) {  
    val dispatcher = activity.onBackPressedDispatcher  
    init {  
        dispatcher.addCallback(object : OnBackPressedCallback(true) {  
            override fun handleOnBackPressedStarted(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressedProgressed(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressed() {...}  
            override fun handleOnBackPressedCancelled() {...}  
        })  
    }  
}
```

OnBackPressedDispatcher

```
class MyComponent(activity: ComponentActivity) {  
    val dispatcher = activity.onBackPressedDispatcher  
    init {  
        dispatcher.addCallback(object : OnBackPressedCallback(true) {  
            override fun handleOnBackPressedStarted(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressedProgressed(backEvent: BackEventCompat) {...}  
            override fun handleOnBackPressed() {...}  
            override fun handleOnBackPressedCancelled() {...}  
        })  
    }  
}
```

OnBackPressedDispatcher

```
class MyComponent(activity: ComponentActivity) {  
    val dispatcher = activity.onBackPressedDispatcher  
    init {  
        dispatcher.addCallback {  
            /* handle system back here */  
        }  
    }  
}
```

OnBackPressedDispatcher

```
class MyComponent(activity: ComponentActivity) {  
    val dispatcher = activity.onBackPressedDispatcher  
    init {  
        dispatcher.addCallback {  
            /* handle system back here */  
        }  
    }  
}
```

PredictiveBackHandler

```
@Composable
fun myComposable() {
    PredictiveBackHandler(true /* enabled condition */) { backEvent ->
        /* handleOnBackStarted */
        try {
            backEvent.collect { /* handleOnBackProgressed */}
            /* handleOnBackPressed */
        } finally {
            /* handleOnBackCancelled */
        }
    }
}
```

PredictiveBackHandler

```
@Composable
fun myComposable() {
    PredictiveBackHandler(true /* enabled condition */) { backEvent ->
        /* handleOnBackStarted */
        try {
            backEvent.collect { /* handleOnBackProgressed */}
            /* handleOnBackPressed */
        } finally {
            /* handleOnBackCancelled */
        }
    }
}
```

PredictiveBackHandler

```
@Composable
fun myComposable() {
    PredictiveBackHandler(true /* enabled condition */) { backEvent ->
        /* handleOnBackStarted */
        try {
            backEvent.collect { /* handleOnBackProgressed */}
            /* handleOnBackPressed */
        } finally {
            /* handleOnBackCancelled */
        }
    }
}
```

PredictiveBackHandler

```
@Composable
fun myComposable() {
    PredictiveBackHandler(true /* enabled condition */) { backEvent ->
        /* handleOnBackStarted */
        try {
            backEvent.collect { /* handleOnBackProgressed */}
            /* handleOnBackPressed */
        } finally {
            /* handleOnBackCancelled */
        }
    }
}
```

PredictiveBackHandler

```
@Composable
fun myComposable() {
    PredictiveBackHandler(true /* enabled condition */) { backEvent ->
        /* handleOnBackStarted */
        try {
            backEvent.collect { /* handleOnBackProgressed */}
            /* handleOnBackPressed */
        } finally {
            /* handleOnBackCancelled */
        }
    }
}
```

PredictiveBackHandler

```
@Composable
fun myComposable() {
    PredictiveBackHandler(true /* enabled condition */) { backEvent ->
        /* handleOnBackStarted */
        try {
            backEvent.collect { /* handleOnBackProgressed */}
            /* handleOnBackPressed */
        } finally {
            /* handleOnBackCancelled */
        }
    }
}
```

PredictiveBackHandler

```
@Composable
fun myComposable() {
    PredictiveBackHandler(true /* enabled condition */) { backEvent ->
        /* handleOnBackStarted */
        try {
            backEvent.collect { /* handleOnBackProgressed */}
            /* handleOnBackPressed */
        } finally {
            /* handleOnBackCancelled */
        }
    }
}
```

PredictiveBackHandler

```
@Composable
fun myComposable() {
    PredictiveBackHandler(true /* enabled condition */) { backEvent ->
        /* handleOnBackStarted */
        try {
            backEvent.collect { /* handleOnBackProgressed */}
            /* handleOnBackPressed */
        } finally {
            /* handleOnBackCancelled */
        }
    }
}
```

PredictiveBackHandler

```
@Composable
fun myComposable() {
    PredictiveBackHandler(true /* enabled condition */) { backEvent ->
        /* handleOnBackStarted */
        try {
            backEvent.collect { /* handleOnBackProgressed */}
            /* handleOnBackPressed */
        } finally {
            /* handleOnBackCancelled */
        }
    }
}
```

Brief Vocabulary

System Back Trio

AndroidX

OnBackPressedDispatcher

onBackPressedCallback

Platform

OnBackInvokedDispatcher

onBackInvokedCallback

AndroidX

NavigationEventDispatcher

NavigationEventCallback

OnBackPressedDispatcher

`Activity.onBackPressed`

`OnBackPressedDispatcher`

`KeyEvent.KEYCODE_BACK`

OnBackPressedDispatcher

`Activity.onBackPressed`

`OnBackPressedDispatcher`

`KeyEvent.KEYCODE_BACK`

OnBackPressedDispatcher

Activity.onBackPressed

OnBackPressedDispatcher

KeyEvent.KEYCODE_BACK

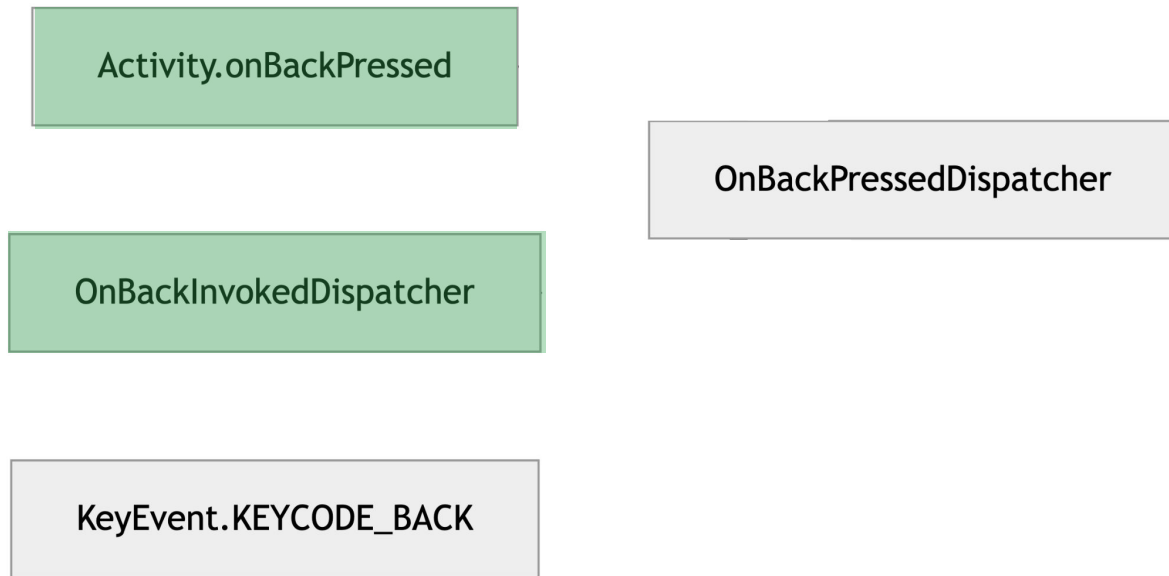
OnBackPressedDispatcher

Activity.onBackPressed

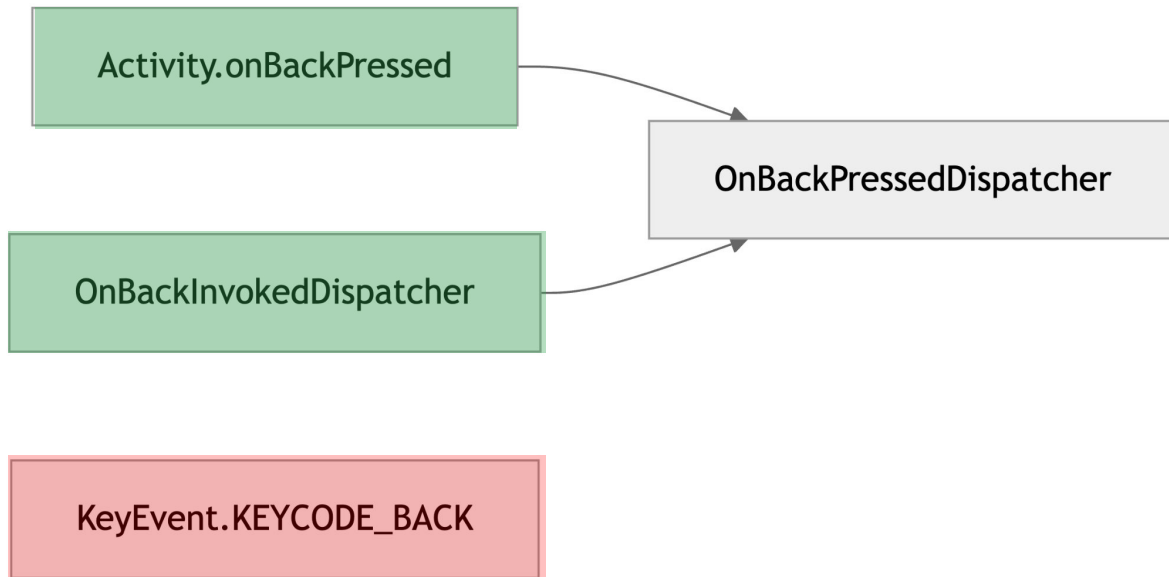
OnBackPressedDispatcher

KeyEvent.KEYCODE_BACK

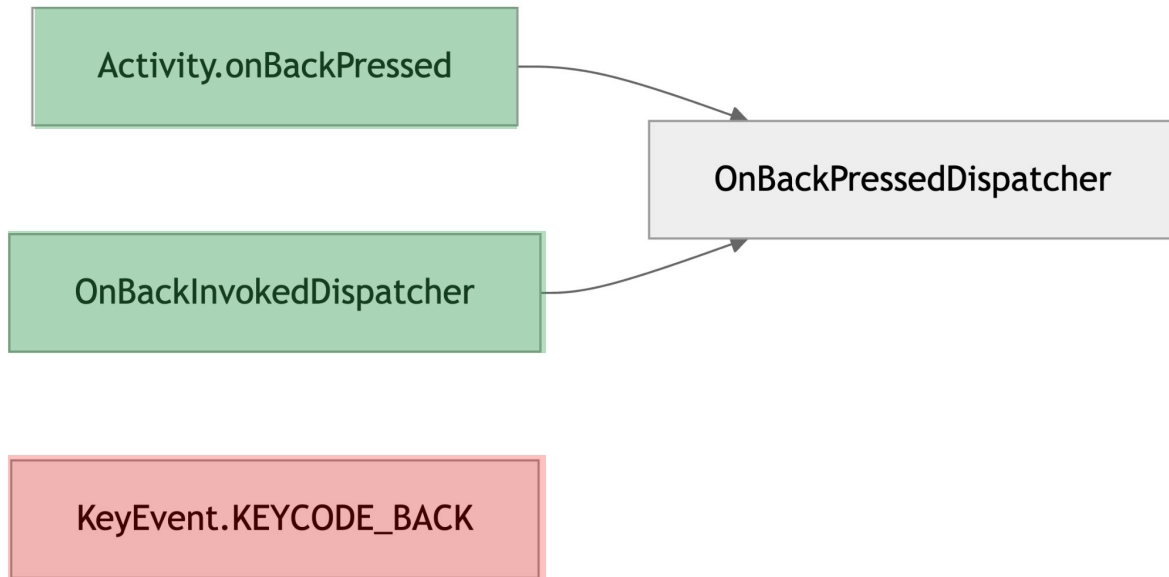
OnBackPressedDispatcher



OnBackPressedDispatcher



OnBackPressedDispatcher

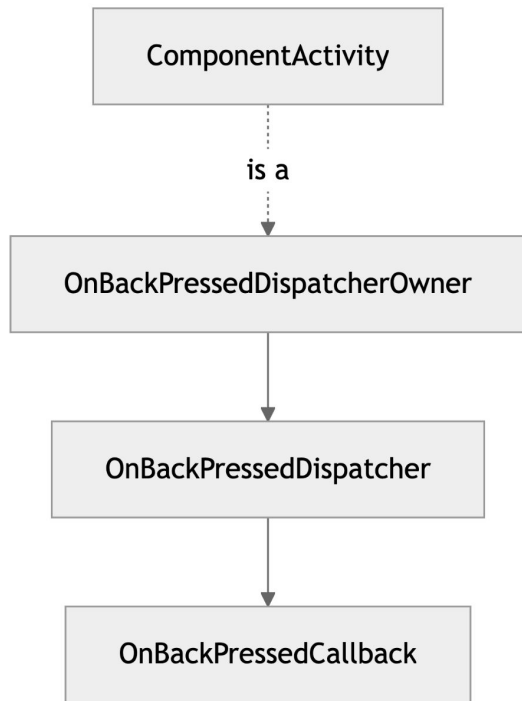


System Events on other Platforms

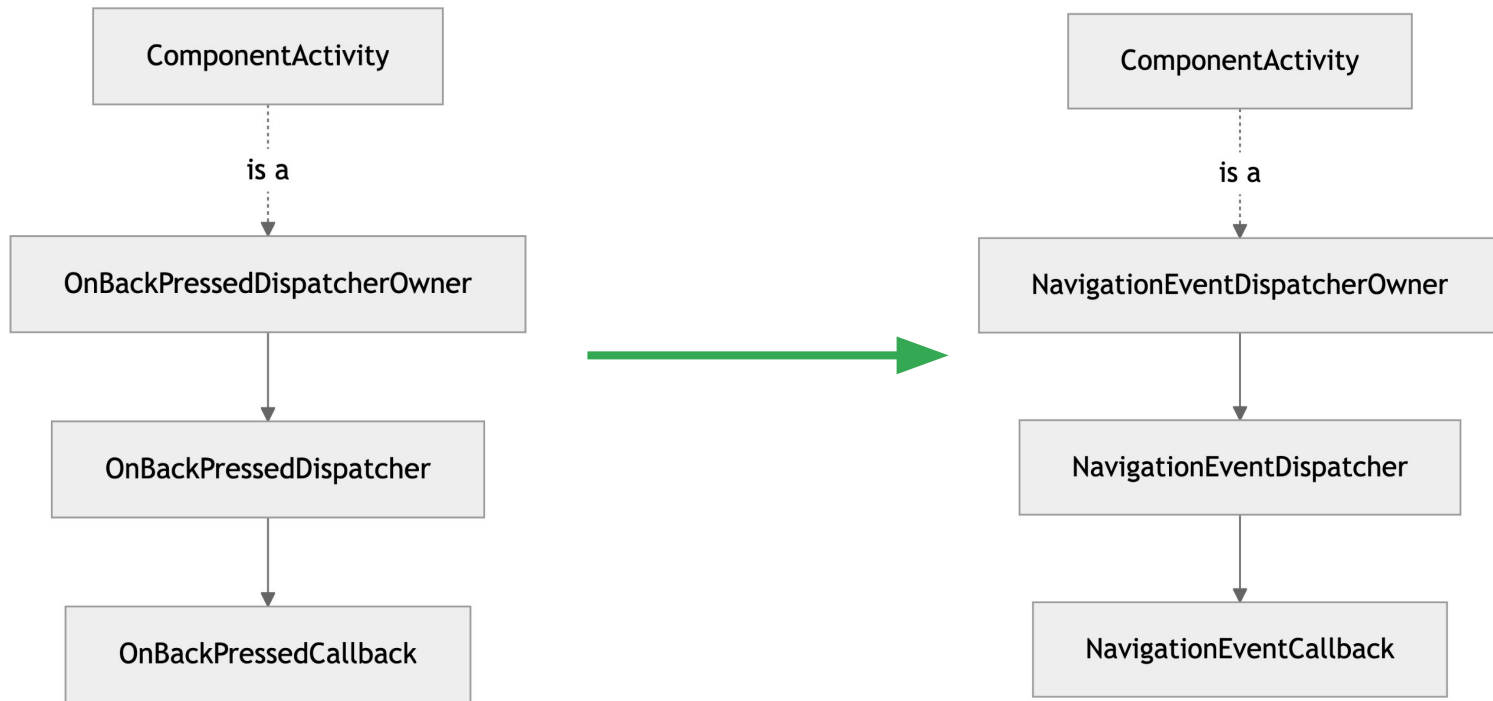
	Back	Up	Forward	Home
Android Phone	✓	✓	?	✓
Android Tablet	✓	✓	?	✓
Android XR	✓	✓	?	✓
Web (Browser)	✓	✓	✓	?
iOS	✓	?	?	✓
Desktop (Multiplatform)	✓	?	?	?

Navigation Event

Navigation Event



Navigation Event



Expect Actual

Expect Classes

```
public expect class NavigationInputHandler {  
    private val dispatcher: NavigationEventDispatcher  
    internal fun updateActiveState()  
}
```

```
public expect class NavigationEvent {  
    public constructor(  
        touchX: Float, touchY: Float, progress: Float,  
        swipeEdge: @SwipeEdge Int, frameTimeMillis: Long = 0  
    )  
}
```

Expect Classes

```
public expect class NavigationInputHandler {  
    private val dispatcher: NavigationEventDispatcher  
    internal fun updateActiveState()  
}  
  
public expect class NavigationEvent {  
    public constructor(  
        touchX: Float, touchY: Float, progress: Float,  
        swipeEdge: @SwipeEdge Int, frameTimeMillis: Long = 0  
    )  
}
```

Expect Classes

```
public expect class NavigationInputHandler {  
    private val dispatcher: NavigationEventDispatcher  
    internal fun updateActiveState()  
}
```

```
public expect class NavigationEvent {  
    public constructor(  
        touchX: Float, touchY: Float, progress: Float,  
        swipeEdge: @SwipeEdge Int, frameTimeMillis: Long = 0  
    )  
}
```

Expect Classes

```
public expect class NavigationInputHandler {  
    private val dispatcher: NavigationEventDispatcher  
    internal fun updateActiveState()  
}
```

```
public expect class NavigationEvent {  
    public constructor(  
        touchX: Float, touchY: Float, progress: Float,  
        swipeEdge: @SwipeEdge Int, frameTimeMillis: Long = 0  
    )  
}
```

Expect Classes

```
public expect class NavigationInputHandler {  
    private val dispatcher: NavigationEventDispatcher  
    internal fun updateActiveState()  
}
```

```
public expect class NavigationEvent {  
    public constructor(  
        touchX: Float, touchY: Float, progress: Float,  
        swipeEdge: @SwipeEdge Int, frameTimeMillis: Long = 0  
    )  
}
```

Expect Classes

```
public expect class NavigationInputHandler {  
    private val dispatcher: NavigationEventDispatcher  
    internal fun updateActiveState()  
}
```

```
public expect class NavigationEvent {  
    public constructor(  
        touchX: Float, touchY: Float, progress: Float,  
        swipeEdge: @SwipeEdge Int, frameTimeMillis: Long = 0  
    )  
}
```

Expect Classes

```
public expect class NavigationInputHandler {  
    private val dispatcher: NavigationEventDispatcher  
    internal fun updateActiveState()  
}
```

```
public expect class NavigationEvent {  
    public constructor(  
        touchX: Float, touchY: Float, progress: Float,  
        swipeEdge: @SwipeEdge Int, frameTimeMillis: Long = 0  
    )  
}
```

Expect Classes

```
public expect class NavigationInputHandler {  
    private val dispatcher: NavigationEventDispatcher  
    internal fun updateActiveState()  
}
```

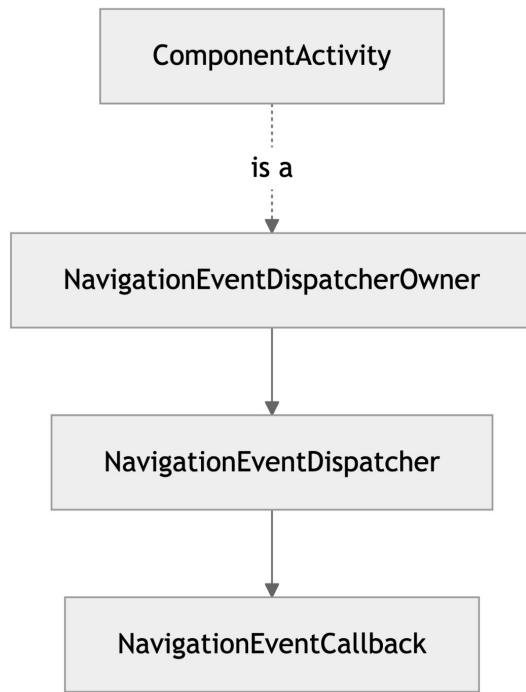
```
public expect class NavigationEvent {  
    public constructor(  
        touchX: Float, touchY: Float, progress: Float,  
        swipeEdge: @SwipeEdge Int, frameTimeMillis: Long = 0  
    )  
}
```

Expect Classes

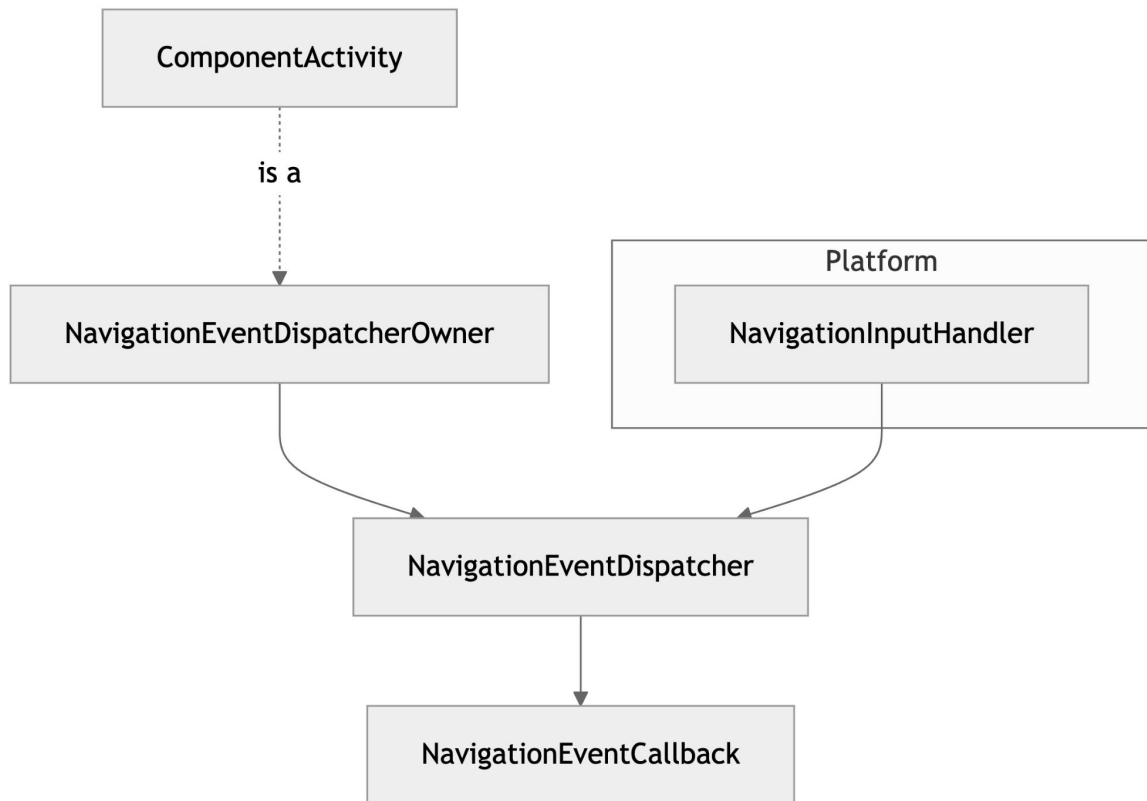
```
public expect class NavigationInputHandler {  
    private val dispatcher: NavigationEventDispatcher  
    internal fun updateActiveState()  
}
```

```
public expect class NavigationEvent {  
    public constructor(  
        touchX: Float, touchY: Float, progress: Float,  
        swipeEdge: @SwipeEdge Int, frameTimeMillis: Long = 0  
    )  
}
```

Navigation Event



Navigation Event



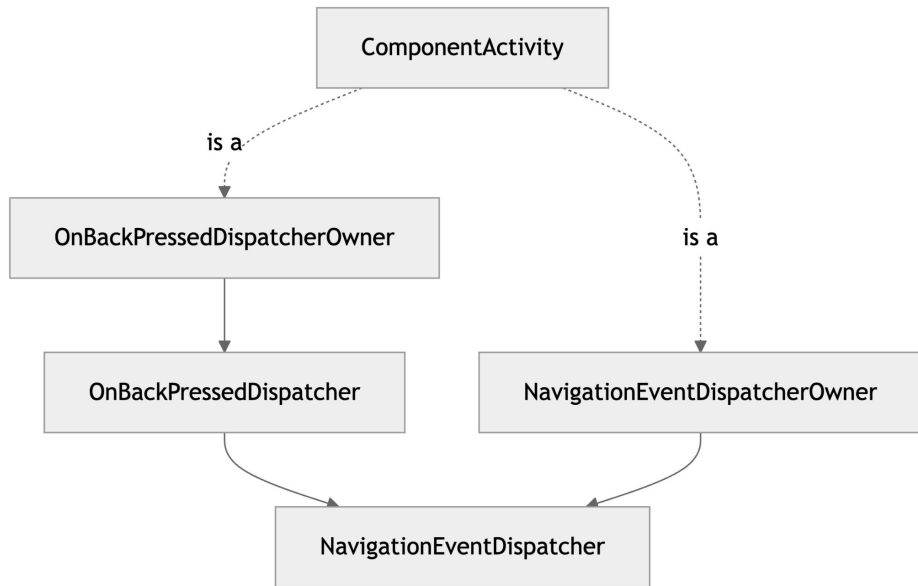
Verifying the new approach

Verifying the new approach

- Re-write OnBackPressedDispatcher on top of
NavigationEvent

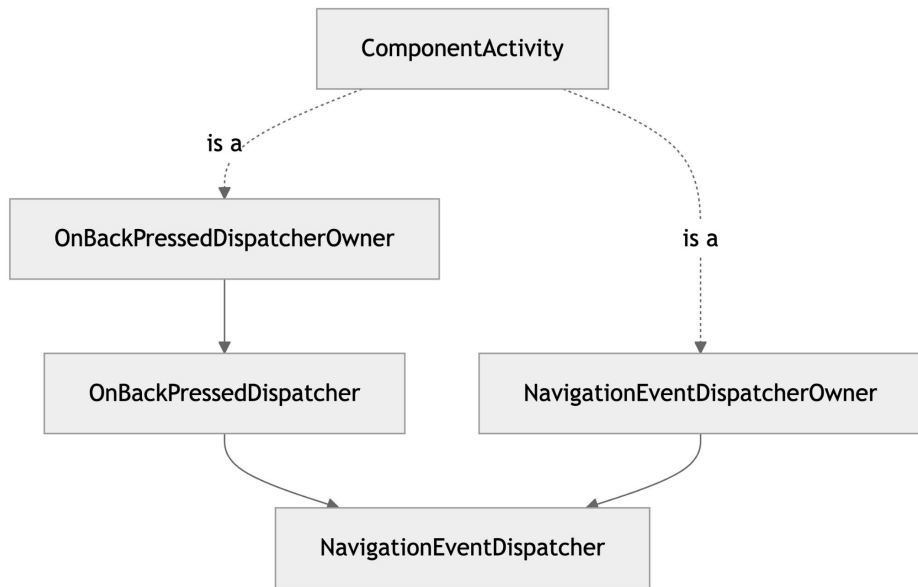
Verifying the new approach

- Re-write OnBackPressedDispatcher on top of NavigationEvent

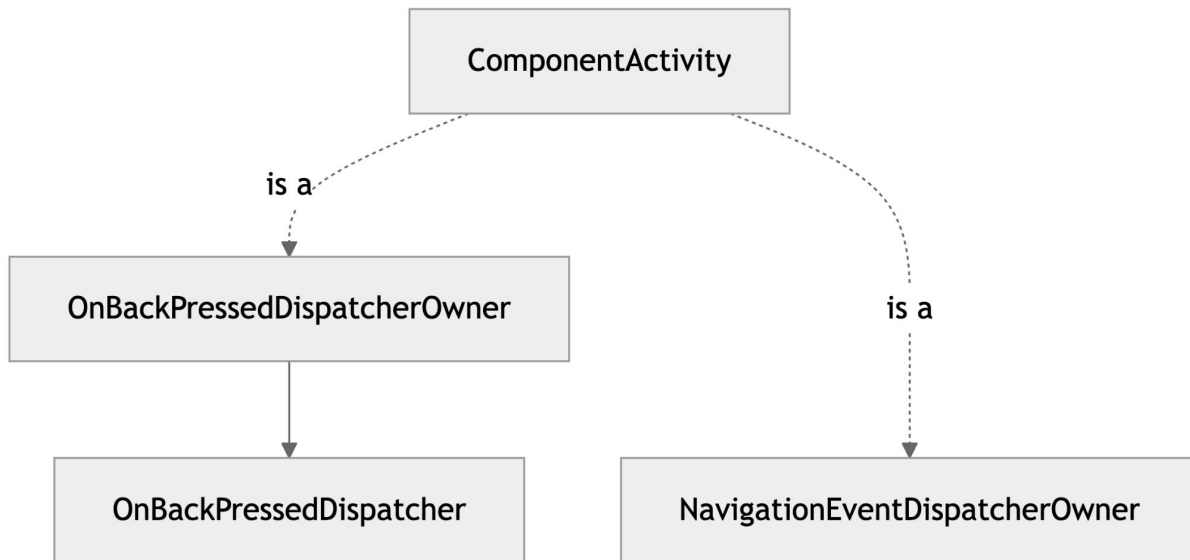


Verifying the new approach

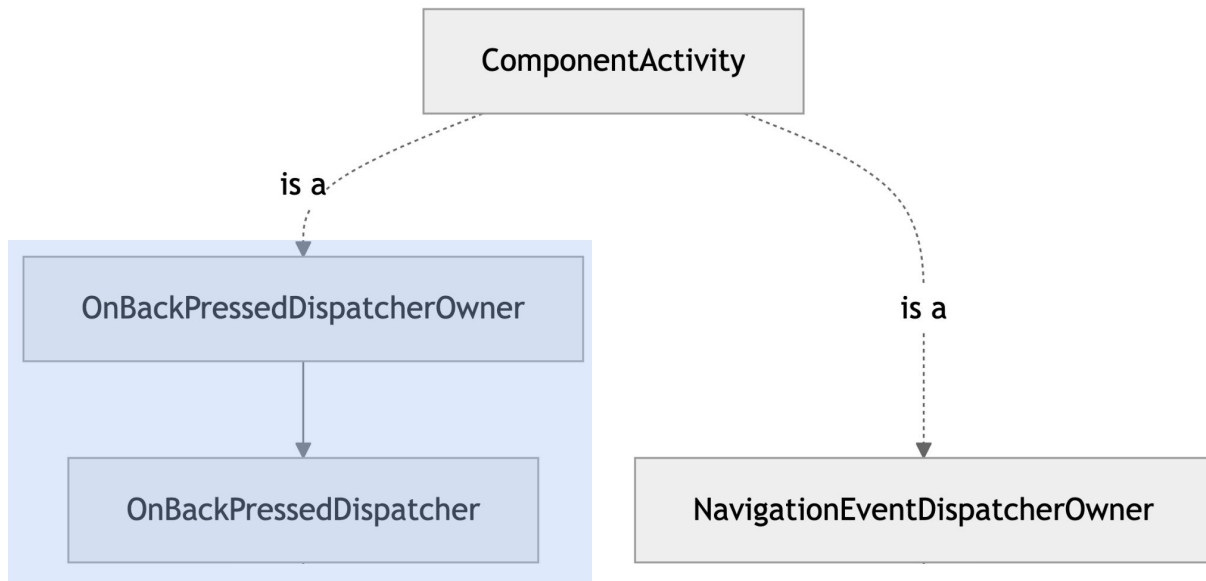
- Re-write OnBackPressedDispatcher on top of NavigationEvent
- Allow integration with the new Navigation3 library



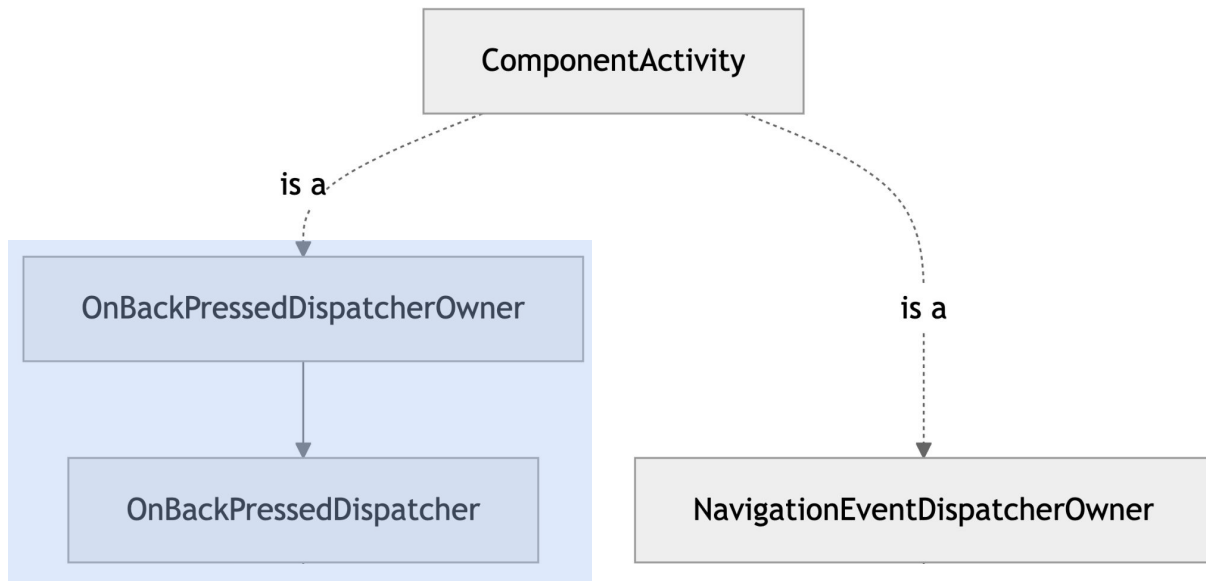
OnBackPressed Rewrite



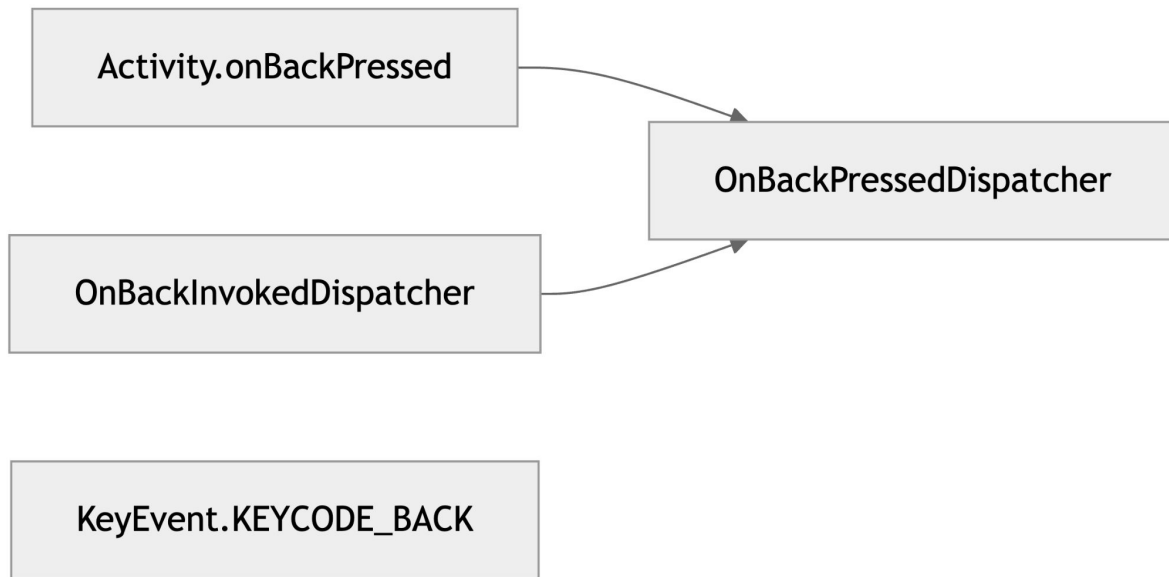
OnBackPressed Rewrite



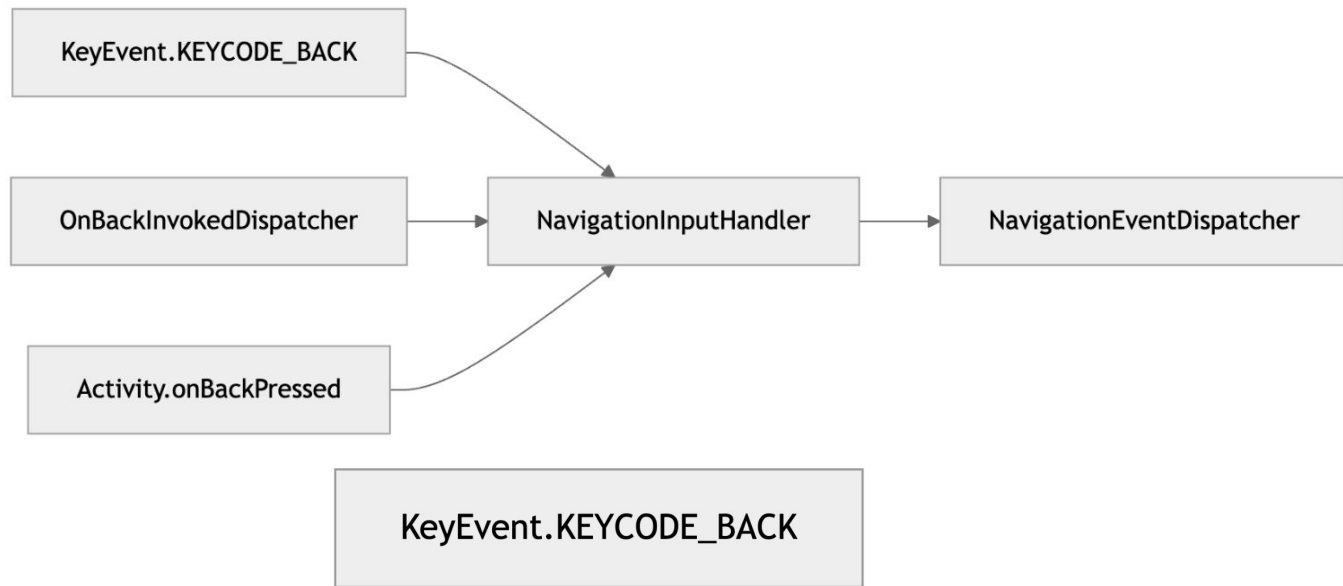
OnBackPressed Rewrite



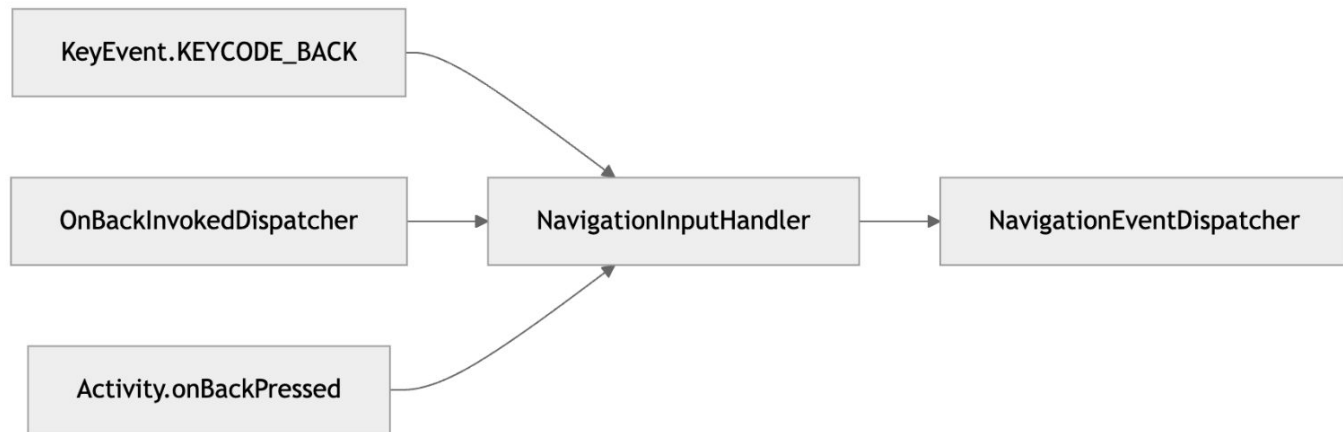
OnBackPressed Rewrite



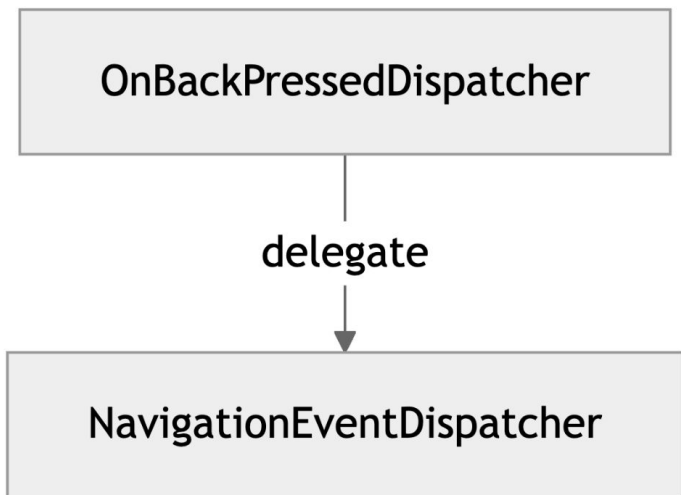
OnBackPressed Rewrite



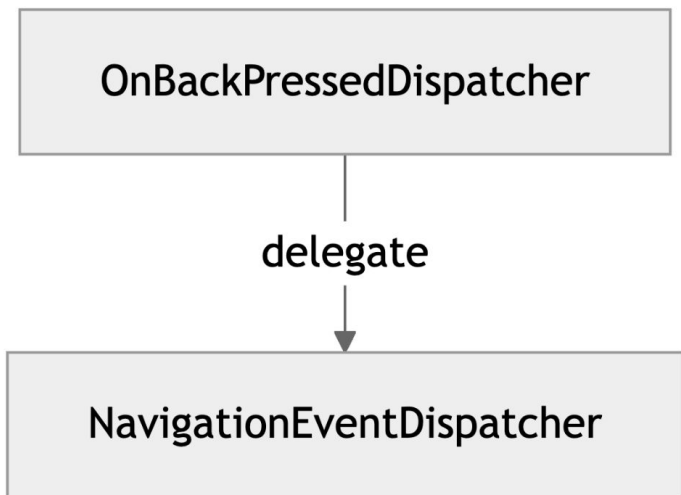
OnBackPressed Rewrite



OnBackPressed Rewrite



OnBackPressed Rewrite



Expect NavigationEvent

```
public expect class NavigationEvent {  
    public constructor(  
        touchX: Float,  
        touchY: Float,  
        progress: Float,  
        swipeEdge: @SwipeEdge Int,  
        frameTimeMillis: Long = 0  
    )  
}
```

Actual NavigationEvent

```
public actual class NavigationEvent actual constructor(...) {  
    public constructor(backEvent: BackEvent) : this(  
        touchX = backEvent.touchX,  
        touchY = backEvent.touchY,  
        progress = backEvent.progress,  
        swipeEdge = backEvent.swipeEdge,  
        frameTimeMillis = backEvent.frameTimeMillis,  
    )  
}
```

Actual NavigationEvent

```
public actual class NavigationEvent actual constructor(...) {  
    public constructor(backEvent: BackEvent) : this(  
        touchX = backEvent.touchX,  
        touchY = backEvent.touchY,  
        progress = backEvent.progress,  
        swipeEdge = backEvent.swipeEdge,  
        frameTimeMillis = backEvent.frameTimeMillis,  
    )  
}
```

Expect NavigationInputHandler

```
public expect class NavigationInputHandler {  
    private val dispatcher: NavigationEventDispatcher  
    internal fun updateActiveState()  
}
```

Actual NavigationInputHandler

```
public actual class NavigationInputHandler(val dispatcher:...) {  
    private var invoker: OnBackInvokedDispatcher? = null  
    private var onBackInvokedCallback: OnBackInvokedCallback? = null  
    private var registered = false  
  
    public fun setOnBackInvokedDispatcher(invoker: OnBackInvokedDispatcher) {  
        this.invoker = invoker  
        updateActiveState()  
    }  
  
    init { ... }  
  
    internal actual fun updateActiveState() { ... }  
}
```

Actual NavigationInputHandler

```
public actual class NavigationInputHandler(val dispatcher:...) {  
    private var invoker: OnBackInvokedDispatcher? = null  
    private var onBackInvokedCallback: OnBackInvokedCallback? = null  
    private var registered = false  
  
    public fun setOnBackInvokedDispatcher(invoker: OnBackInvokedDispatcher) {  
        this.invoker = invoker  
        updateActiveState()  
    }  
  
    init { ... }  
  
    internal actual fun updateActiveState() { ... }  
}
```

Actual NavigationInputHandler

```
public actual class NavigationInputHandler(val dispatcher:...) {  
    private var invoker: OnBackInvokedDispatcher? = null  
    private var onBackInvokedCallback: OnBackInvokedCallback? = null  
    private var registered = false  
  
    public fun setOnBackInvokedDispatcher(invoker: OnBackInvokedDispatcher) {  
        this.invoker = invoker  
        updateActiveState()  
    }  
  
    init { ... }  
  
    internal actual fun updateActiveState() { ... }  
}
```

Actual NavigationInputHandler

```
public actual class NavigationInputHandler(val dispatcher:...) {  
    private var invoker: OnBackInvokedDispatcher? = null  
    private var onBackInvokedCallback: OnBackInvokedCallback? = null  
    private var registered = false  
  
    public fun setOnBackInvokedDispatcher(invoker: OnBackInvokedDispatcher) {  
        this.invoker = invoker  
        updateActiveState()  
    }  
  
    init { ... }  
  
    internal actual fun updateActiveState() { ... }  
}
```

Actual NavigationInputHandler

```
public actual class NavigationInputHandler(val dispatcher:...) {  
    private var invoker: OnBackInvokedDispatcher? = null  
    private var onBackInvokedCallback: OnBackInvokedCallback? = null  
    private var registered = false  
  
    public fun setOnBackInvokedDispatcher(invoker: OnBackInvokedDispatcher) {  
        this.invoker = invoker  
        updateActiveState()  
    }  
  
    init { ... }  
  
    internal actual fun updateActiveState() { ... }  
}
```

Actual NavigationInputHandler

```
public actual class NavigationInputHandler(val dispatcher:...) {  
    private var invoker: OnBackInvokedDispatcher? = null  
    private var onBackInvokedCallback: OnBackInvokedCallback? = null  
    private var registered = false  
  
    public fun setOnBackInvokedDispatcher(invoker: OnBackInvokedDispatcher) {  
        this.invoker = invoker  
        updateActiveState()  
    }  
  
    init { ... }  
  
    internal actual fun updateActiveState() { ... }  
}
```

Actual NavigationInputHandler

```
public actual class NavigationInputHandler(val dispatcher:...) {  
    private var invoker: OnBackInvokedDispatcher? = null  
    private var onBackInvokedCallback: OnBackInvokedCallback? = null  
    private var registered = false  
  
    public fun setOnBackInvokedDispatcher(invoker: OnBackInvokedDispatcher) {  
        this.invoker = invoker  
        updateActiveState()  
    }  
  
    init { ... }  
  
    internal actual fun updateActiveState() { ... }  
}
```

Actual NavigationInputHandler

```
public actual class NavigationInputHandler(val dispatcher:...) {  
    private var invoker: OnBackInvokedDispatcher? = null  
    private var onBackInvokedCallback: OnBackInvokedCallback? = null  
    private var registered = false  
  
    public fun setOnBackInvokedDispatcher(invoker: OnBackInvokedDispatcher) {  
        this.invoker = invoker  
        updateActiveState()  
    }  
  
    init { ... }  
  
    internal actual fun updateActiveState() { ... }  
}
```

Actual NavigationInputHandler

```
internal actual fun updateActiveState() {  
    val shouldBeRegistered = dispatcher.hasEnabledCallbacks()\br/>    if (shouldBeRegistered && !registered) {  
        registerOnBackPressedCallback(...)  
    } else if (!shouldBeRegistered && registered) {  
        unregisterOnBackPressedCallback(...)  
    }  
}
```

Actual NavigationInputHandler

```
internal actual fun updateActiveState() {  
    val shouldBeRegistered = dispatcher.hasEnabledCallbacks()  
    if (shouldBeRegistered && !registered) {  
        registerOnBackPressedCallback(...)  
    } else if (!shouldBeRegistered && registered) {  
        unregisterOnBackPressedCallback(...)  
    }  
}
```

Actual NavigationInputHandler

```
internal actual fun updateActiveState() {  
    val shouldBeRegistered = dispatcher.hasEnabledCallbacks()  
    if (shouldBeRegistered && !registered) {  
        registerOnBackInvokedCallback(...)  
    } else if (!shouldBeRegistered && registered) {  
        unregisterOnBackInvokedCallback(...)  
    }  
}
```

Actual NavigationInputHandler

```
public actual class NavigationInputHandler(val dispatcher:...) {  
    private var invoker: OnBackInvokedDispatcher? = null  
    private var onBackInvokedCallback: OnBackInvokedCallback? = null  
    private var registered = false  
  
    public fun setOnBackInvokedDispatcher(invoker: OnBackInvokedDispatcher) {  
        this.invoker = invoker  
        updateActiveState()  
    }  
  
    init { ... }  
  
    internal actual fun updateActiveState() { ... }  
}
```

Actual NavigationInputHandler

```
public class NavigationInputHandler(val dispatcher:...) {  
    private var onBackInvokedCallback: OnBackInvokedCallback? = null  
    init {  
        onBackInvokedCallback: OnBackInvokedCallback = OnBackAnimationCallback {  
            override fun onBackStarted(backEvent: BackEvent) {  
                dispatcher.dispatchOnStarted(NavigationEvent(backEvent))  
            }  
            override fun onBackProgressed(backEvent: BackEvent) {  
                dispatcher.dispatchOnProgressed(NavigationEvent(backEvent))  
            }  
            override fun onBackInvoked() { dispatcher.dispatchOnCompleted() }  
            override fun onBackCancelled() { dispatcher.dispatchOnCancelled() }  
        }  
    }  
}
```

Actual NavigationInputHandler

```
public class NavigationInputHandler(val dispatcher:...) {  
    private var onBackInvokedCallback: OnBackInvokedCallback? = null  
    init {  
        onBackInvokedCallback: OnBackInvokedCallback = OnBackAnimationCallback {  
            override fun onBackStarted(backEvent: BackEvent) {  
                dispatcher.dispatchOnStarted(NavigationEvent(backEvent))  
            }  
            override fun onBackProgressed(backEvent: BackEvent) {  
                dispatcher.dispatchOnProgressed(NavigationEvent(backEvent))  
            }  
            override fun onBackInvoked() { dispatcher.dispatchOnCompleted() }  
            override fun onBackCancelled() { dispatcher.dispatchOnCancelled() }  
        }  
    }  
}
```

Actual NavigationInputHandler

```
public class NavigationInputHandler(val dispatcher:...) {  
    private var onBackInvokedCallback: OnBackInvokedCallback? = null  
    init {  
        onBackInvokedCallback: OnBackInvokedCallback = OnBackAnimationCallback {  
            override fun onBackStarted(backEvent: BackEvent) {  
                dispatcher.dispatchOnStarted(NavigationEvent(backEvent))  
            }  
            override fun onBackProgressed(backEvent: BackEvent) {  
                dispatcher.dispatchOnProgressed(NavigationEvent(backEvent))  
            }  
            override fun onBackInvoked() { dispatcher.dispatchOnCompleted() }  
            override fun onBackCancelled() { dispatcher.dispatchOnCancelled() }  
        }  
    }  
}
```

OnBackPressedCallback

```
abstract class OnBackPressedCallback(enabled: Boolean) {  
    internal val eventCallback =  
        object : NavigationEventCallback(isEnabled = enabled) {  
            override fun onEventStarted(event: NavigationEvent) {  
                handleOnBackPressed(BackEventCompat(event))  
            }  
            override fun onEventProgressed(event: NavigationEvent) {  
                handleOnBackPressed(BackEventCompat(event))  
            }  
            ...  
        }  
}
```

OnBackPressedCallback

```
abstract class OnBackPressedCallback(enabled: Boolean) {  
    internal val eventCallback =  
        object : NavigationEventCallback(isEnabled = enabled) {  
            override fun onEventStarted(event: NavigationEvent) {  
                handleOnBackPressed(BackEventCompat(event))  
            }  
            override fun onEventProgressed(event: NavigationEvent) {  
                handleOnBackPressed(BackEventCompat(event))  
            }  
            ...  
        }  
}
```

OnBackPressedDispatcher

```
class OnBackPressedDispatcher(...) {  
    fun addCallback(onBackPressedCallback: OnBackPressedCallback) {  
        eventDispatcher.addCallback(onBackPressedCallback.eventCallback)  
    }  
    private fun onBackPressed(backEvent: BackEventCompat) {  
        eventDispatcher.dispatchOnStarted(backEvent.toNavigationEvent())  
    }  
    private fun onBackPressed(backEvent: BackEventCompat) {  
        eventDispatcher.dispatchOnProgressed(backEvent.toNavigationEvent())  
    }  
    ...  
}
```

OnBackPressedDispatcher

```
class OnBackPressedDispatcher(...) {  
    fun addCallback(onBackPressedCallback: OnBackPressedCallback) {  
        eventDispatcher.addCallback(onBackPressedCallback.eventCallback)  
    }  
    private fun onBackStarted(backEvent: BackEventCompat) {  
        eventDispatcher.dispatchOnStarted(backEvent.toNavigationEvent())  
    }  
    private fun onBackProgressed(backEvent: BackEventCompat) {  
        eventDispatcher.dispatchOnProgressed(backEvent.toNavigationEvent())  
    }  
    ...  
}
```

OnBackPressedDispatcher

```
class OnBackPressedDispatcher(...) {  
    fun addCallback(onBackPressedCallback: OnBackPressedCallback) {  
        eventDispatcher.addCallback(onBackPressedCallback.eventCallback)  
    }  
    private fun onBackStarted(backEvent: BackEventCompat) {  
        eventDispatcher.dispatchOnStarted(backEvent.toNavigationEvent())  
    }  
    private fun onBackProgressed(backEvent: BackEventCompat) {  
        eventDispatcher.dispatchOnProgressed(backEvent.toNavigationEvent())  
    }  
    ...  
}
```

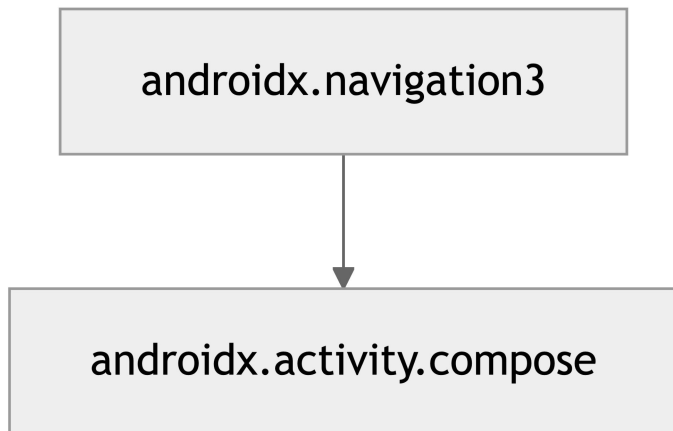
OnBackPressedDispatcher

```
class OnBackPressedDispatcher(...) {  
    fun addCallback(onBackPressedCallback: OnBackPressedCallback) {  
        eventDispatcher.addCallback(onBackPressedCallback.eventCallback)  
    }  
    private fun onBackPressed(backEvent: BackEventCompat) {  
        eventDispatcher.dispatchOnStarted(backEvent.toNavigationEvent())  
    }  
    private fun onBackPressed(backEvent: BackEventCompat) {  
        eventDispatcher.dispatchOnProgressed(backEvent.toNavigationEvent())  
    }  
    ...  
}
```

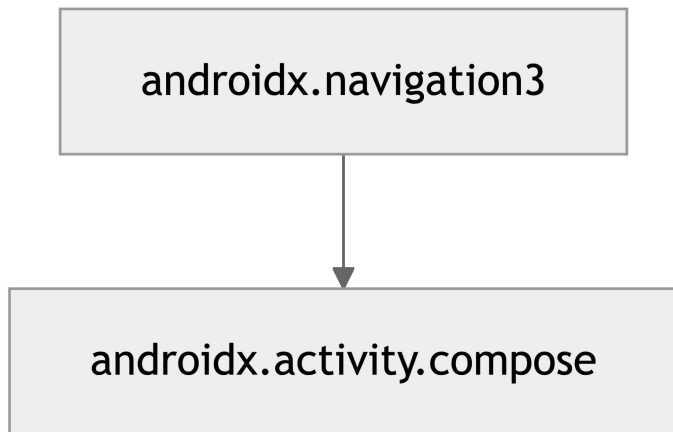
OnBackPressedDispatcher

```
class OnBackPressedDispatcher(...) {  
    fun addCallback(  
        owner: LifecycleOwner, onBackPressedCallback: OnBackPressedCallback  
    ) {  
        ...  
    }  
    ...  
}
```

Navigation 3 Integration



Navigation 3 Integration



Navigation 3 Integration



Navigation 3 Integration



What's next for Navigation Event?



What's next for Navigation Event?

● System Back

○ Predictive Back



What's next for Navigation Event?

- ~~System Back~~
- ~~Predictive Back~~
- Keyboard Events



What's next for Navigation Event?

- ~~System Back~~
- ~~Predictive Back~~
- Keyboard Events
- Forward Navigation



04

Review



Migrate a Library

✓ Keeps existing behavior and code that's already tested.

✓ Can move to KMP step-by-step, reducing risk.

✓ Customers don't need to change their code.

✗ Harder to clean up old or platform-specific design.

✗ May involve complex refactoring and workarounds.

Migrate a Library

- ✓ Keeps existing behavior and code that's already tested.
- ✓ Can move to KMP step-by-step, reducing risk.
- ✓ Customers don't need to change their code.
- ✗ Harder to clean up old or platform-specific design.
- ✗ May involve complex refactoring and workarounds.

Rewrite a Library

- ✓ Clean design from the start, no legacy "platform-specific" code.
- ✓ Easier to apply modern best practices.
- ✗ Slower to build feature parity if the existing library is large.
- ✗ May miss edge cases already handled in the existing code.
- ✗ Customers will need to migrate.

Migrate a Library

- ✓ Keeps existing behavior and code that's already tested.
- ✓ Can move to KMP step-by-step, reducing risk.
- ✓ Customers don't need to change their code.
- ✗ Harder to clean up old or platform-specific design.
- ✗ May involve complex refactoring and workarounds.

Rewrite a Library

- ✓ Clean design from the start, no legacy "platform-specific" code.
- ✓ Easier to apply modern best practices.
- ✗ Slower to build feature parity if the existing library is large.
- ✗ May miss edge cases already handled in the existing code.
- ✗ Customers will need to migrate.



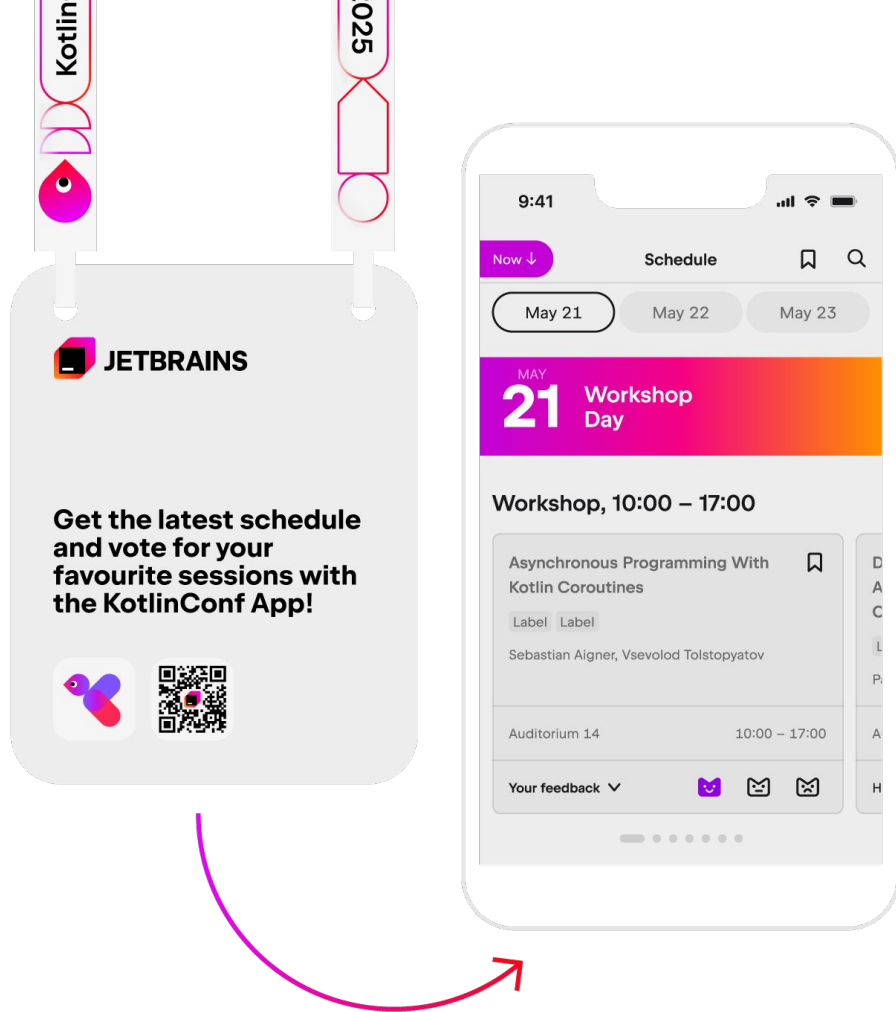
New Releases!

- SavedState 1.3 (KMP + Serialization)
- NavigationEvent Alpha01
- Navigation3 Alpha01



goo.gle/nav3

Thank You, and Don't Forget to Vote



```

if (action == Action.Back) {
    val event = NavigationEvent(currentX, currentY, calculateProgress(startX, currentX, windowSize.width.toFloat()), Edge.Left)
    backChannel.onProgressed(event)
}

override fun dispatchEvent(e: NavigationEventWrapper, callback: NavigationEventCallback) {
    when (e.action) {
        Action.Back -> {
            when (e.type) {
                EventType.Started -> {
                    backNavigation?.let {
                        it.callback.onBackCancelled()
                    }
                    backNavigation = Navigation(callback.deepCopy()).also {
                        it.callback.onBackStarted(e.externalEvent!!)
                    }
                }
                EventType.Progressed -> backNavigation?.callback?.onBackProgressed(e.externalEvent!!)
                EventType.Cancelled -> {
                    backNavigation?.callback?.onBackCancelled()
                    backNavigation = null
                }
                EventType.Completed -> {
                    backNavigation?.callback?.onBackCompleted()
                    backNavigation = null
                }
            }
        }
    }
}

```