



# PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 2

## WYKŁAD 6

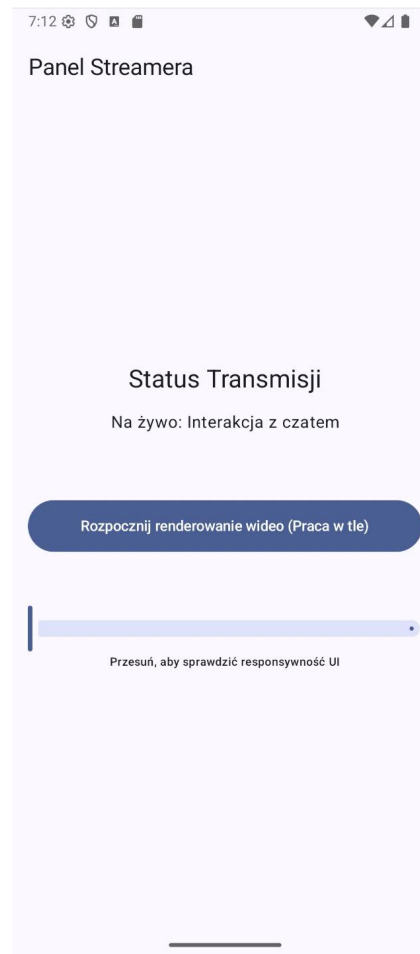
Zaawansowane Zarządzanie Stanem:

- withContext
- StateIn, SharedIn
- FlowOn, combine

Głównym narzędziem do **bezpiecznego** wykonywania operacji w tle jest **withContext**. Pozwala on na **przeniesienie bloku kodu** na inny wątek (dispatcher) bez łamania struktury korutyny.

Głównym narzędziem do **bezpiecznego** wykonywania operacji w tle jest **withContext**. Pozwala on na **przeniesienie bloku kodu** na inny wątek (dispatcher) bez łamania struktury korutyny.

**Zawiesza** korutynę na bieżącym wątku (np. Main), wykonuje pracę na wątku w tle (np. Dispatchers.Default dla obliczeń lub Dispatchers.IO dla operacji plikowych/sieciowych), a po jej zakończeniu wznawia korutynę na **pierwotnym wątku** z gotowym wynikiem.



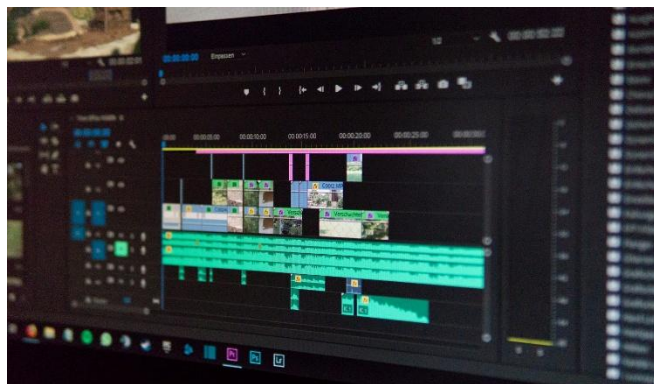
Założmy że nasz Streamer prowadzi transmisję na żywo.

Jego **głównym zadaniem** jest interakcja z czatem, granie i komentowanie – wszystko to dzieje się w **czasie rzeczywistym** i musi być **płynne**. To jest jego praca na **Dispatchers.Main** (**wątek UI**).



Założmy że nasz Streamer prowadzi transmisję na żywo.

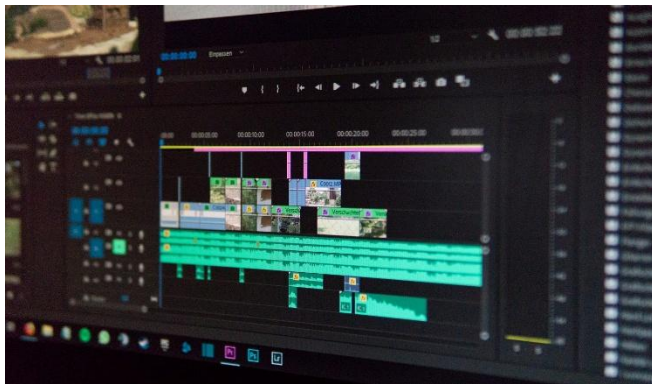
Jego **głównym zadaniem** jest interakcja z czatem, granie i komentowanie – wszystko to dzieje się w **czasie rzeczywistym** i musi być **płynne**. To jest jego praca na `Dispatchers.Main` (**wątek UI**).



Streamer decyduje, że chce wyrenderować film 4K na swój kanał YouTube. Gdyby spróbował to zrobić na tym samym komputerze, na którym prowadzi transmisję, jego **stream** natychmiast by się **zwiesił**. Rozwiązaniem tego problemu jest zastosowanie **withContext**.


Założmy że nasz Streamer prowadzi transmisję na żywo.

Jego **głównym zadaniem** jest interakcja z czatem, granie i komentowanie – wszystko to dzieje się w **czasie rzeczywistym** i musi być **płynne**. To jest jego praca na **Dispatchers.Main (wątek UI)**.



Streamer decyduje, że chce wyrenderować film 4K na swój kanał YouTube. Gdyby spróbował to zrobić na tym samym komputerze, na którym prowadzi transmisję, jego **stream** natychmiast by się **zwiesił**. Rozwiązaniem tego problemu jest zastosowanie **withContext**.

Używa **withContext**, co jest jak wysłanie wszystkich plików na inny komputer, który jest zoptymalizowany pod intensywne obliczenia. Streamer może dalej bez problemu prowadzić transmisję na żywo (**UI jest responsywne**). Gdy komputer skończy renderować film, odsyła gotowy plik z powrotem. Korutyna jest wznowiana na **Dispatchers.Main** z gotowym wynikiem.

7:12 

Panel Streamera

Status Transmisji

Na żywo: Interakcja z czatem

Rozpocznij renderowanie wideo (Praca w tle)

Przesuń, aby sprawdzić responsywność UI

```
data class StreamerUiState(  
    val isLoading: Boolean = false,  
    val status: String = "Na żywo: Interakcja z czatem"  
)
```

1 Usage

```
class StreamerViewModel : ViewModel() {  
    3 Usages  
    private val _uiState = MutableStateFlow( value = StreamerUiState())  
    1 Usage  
    val uiState: StateFlow<StreamerUiState> = _uiState.asStateFlow()  
  
    1 Usage  
    fun startVideoRender() {...}  
  
    1 Usage  
    private suspend fun simulateHeavyCpuRender(): String {...}  
}
```

Model stanu UI - czy jest ładowanie, jaki jest tekst statusu

ViewModel przechowuje cały stan UI w jednym obiekcie StreamerUiState zarządzanym przez StateFlow

```
data class StreamerUiState(  
    val isLoading: Boolean = false,  
    val status: String = "Na żywo: Interakcja z czatem"  
)  
  
1 Usage  
class StreamerViewModel : ViewModel() {  
    3 Usages  
    private val _uiState = MutableStateFlow(value = StreamerUiState())  
    1 Usage  
    val uiState: StateFlow<StreamerUiState> = _uiState.asStateFlow()  
  
    1 Usage  
    fun startVideoRender() {...}  
  
    1 Usage  
    private suspend fun simulateHeavyCpuRender(): String {...}  
}
```



Model stanu UI - czy jest ładowanie, jaki jest tekst statusu

ViewModel przechowuje cały stan UI w jednym obiekcie StreamerUiState zarządzanym przez StateFlow

punkt startowy operacji asynchronicznej

Funkcja symulująca „ciężkie” obliczenia

```
data class StreamerUiState(  
    val isLoading: Boolean = false,  
    val status: String = "Na żywo: Interakcja z czatem"  
)  
  
1 Usage  
class StreamerViewModel : ViewModel() {  
    3 Usages  
    private val _uiState = MutableStateFlow(value = StreamerUiState())  
    1 Usage  
    val uiState: StateFlow<StreamerUiState> = _uiState.asStateFlow()  
  
    1 Usage  
    fun startVideoRender() {...}  
  
    1 Usage  
    private suspend fun simulateHeavyCpuRender(): String {...}  
}
```

Model stanu UI - czy jest ładowanie, jaki jest tekst statusu

ViewModel przechowuje cały stan UI w jednym obiekcie StreamerUiState zarządzanym przez StateFlow

punkt startowy operacji asynchronicznej

Funkcja symulująca „ciężkie” obliczenia


**Nie musi** być oznaczony jako `suspend`, ponieważ wewnątrz nie wywołuje żadnej innej funkcji `suspend`. Jednak oznaczenie jest **dobrą i zalecaną** praktyką: **Sygnalizowanie intencji**, oznaczając funkcję jako `suspend`, **zmuszasz** każdego, kto chce jej użyć, do zrobienia tego **wewnątrz korutyny**.

```
data class StreamerUiState(  
    val isLoading: Boolean = false,  
    val status: String = "Na żywo: Interakcja z czatem"  
)  
  
1 Usage  
class StreamerViewModel : ViewModel() {  
    3 Usages  
    private val _uiState = MutableStateFlow(value = StreamerUiState())  
    1 Usage  
    val uiState: StateFlow<StreamerUiState> = _uiState.asStateFlow()  
  
    1 Usage  
    fun startVideoRender() {...}  
  
    1 Usage  
    private suspend fun simulateHeavyCpuRender(): String {...}  
}
```

```
private suspend fun simulateHeavyCpuRender(): String {  
    var progress = 0  
    for (i in 1..100) {  
        Thread.sleep(millis = 30)  
        progress = i  
    }  
    return "Renderowanie 4K zakończone! ($progress%)"  
}
```

# withContext

**uruchamia** korutynę, która jest **bezpiecznie powiązana** z cyklem życia ViewModelu. Domyślnie startuje na **głównym wątku UI (Dispatchers.Main)**, co pozwala na natychmiastową zmianę stanu (np. ustawienie `isLoading = true`) przed zleceniem ciężkiej pracy.



```
fun startVideoRender() {  
    viewModelScope.launch {  
        _uiState.update {  
            it.copy(isLoading = true,  
                status = "Wysyłam pliki do serwerowni...")  
        }  
  
        val result = withContext(context = Dispatchers.Default) {  
            simulateHeavyCpuRender()  
        }  
  
        _uiState.update { it.copy(isLoading = false, status = result) }  
    }  
}
```

# withContext

**uruchamia** korutynę, która jest **bezpiecznie powiązana** z cyklem życia ViewModelu. Domyślnie startuje na **głównym wątku UI (Dispatchers.Main)**, co pozwala na natychmiastową zmianę stanu (np. ustawienie `isLoading = true`) przed zleceniem ciężkiej pracy.

```
fun startVideoRender() {  
    viewModelScope.launch {  
        _uiState.update {  
            it.copy(isLoading = true,  
                status = "Wysyłam pliki do serwerowni...")  
        }  
    }  
}
```

`withContext (Dispatchers.Default)` **zawiesza** korutynę na wątku głównym i **przenosi** jej wykonanie na wątek z puli `Dispatchers.Default` (**przełączenie kontekstu**). Po zakończeniu bloku kodu, korutyna **automatycznie** wznowia działanie na **wątku głównym** z gotowym wynikiem (`result`).

```
val result = withContext(context = Dispatchers.Default) {  
    simulateHeavyCpuRender()  
}  
  
_uiState.update { it.copy(isLoading = false, status = result) }
```

11:27

## Demo withContext

### Test Responsywności UI



Przesuń, aby sprawdzić, czy UI odpowiada

Gotowy do pracy.

✗ Uruchom Blokująco (BŁĄD)

✓ Uruchom Poprawnie (z withContext)

1 Usage

```
fun runBlockingCalculation() {
    viewModelScope.launch { // Startuje na Dispatchers.Main
        _uiState.value = UiState(
            isLoading = true,
            result = "Obliczenia blokujące START..."
        )
        val calculationResult = performHeavyCalculation()
        _uiState.value = UiState(
            isLoading = false,
            result = "Blokujące ZAKOŃCZONE: $calculationResult"
        )
    }
}
```

1 Usage

```
fun runCorrectCalculation() {
    viewModelScope.launch { // Startuje na Dispatchers.Main
        _uiState.value = UiState(
            isLoading = true,
            result = "Obliczenia w tle START..."
        )
        val calculationResult = withContext(context = Dispatchers.Default) {
            performHeavyCalculation()
        }
        _uiState.value = UiState(
            isLoading = false,
            result = "Poprawne ZAKOŃCZONE: $calculationResult"
        )
    }
}
```

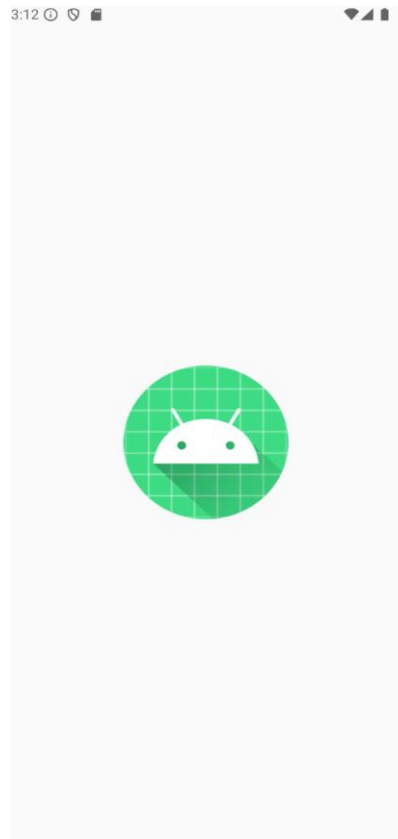
# stateIn vs shareIn

**stateIn** i **shareIn** to operatory, które zamieniają **zimny** Flow w **gorący** strumień.

- **stateIn** – **wymaga** wartości **początkowej**, **zawsze** trzyma **ostatnią wartość**. Nowy subskrybent **od razu** otrzymuje **ostatnią wartość**. Zastosowanie do stanu.
- **shareIn** – **nie ma** stanu początkowego. **Opcjonalnie** buforuje **ostatnią wartość**. Zastosowanie: Jednorazowe lub wielokrotne **zdarzenia**.

Wyobraźmy sobie, że nasz streamer ma na dysku dwa gotowe, zmontowane pliki wideo (**zimny Flow**).

- Nagranie "*Top 5 Klipów Tygodnia*". To jest materiał, który **ma stan** – **zawsze** na ekranie jest **widoczny jakiś klip**.
- Ścieżka dźwiękowa z alertami (dźwięk subskrypcji, dźwięk donacji), które mają się **pojawić w określonych momentach**. To są **zdarzenia**.



# stateIn vs shareIn

**Zimny strumień** gotowy do uruchomienia. Zawiera listę, przez którą przechodzi i emituje element co 3 sekundy

// 1. ZIMNY STRUMIEŃ: Gotowy plik wideo "Top 5 Klipów"

1 Usage

```
private val topPlaysVideo: Flow<String> = flow {  
    val clips = listOf("Klip #1: Stream 3!", "Klip #2: Stream 1!",  
        "Klip #3: Stream 5!", "Klip #4: Stream 12!")  
    for (clip in clips) {  
        emit(value = clip)  
        delay(timeMillis = 3000)  
    }  
}
```

// 2. GORĄCY STATEFLOW: "Kanał Twitch" retransmitujący wideo na żywo

1 Usage

```
val liveTopPlays: StateFlow<String> = topPlaysVideo  
    .stateIn(  
        scope = viewModelScope,  
        started = SharingStarted.WhileSubscribed(stopTimeoutMillis = 5000),  
        initialValue = "Retransmisja rozpocznie się za chwilę..."  
    )
```



# stateIn vs shareIn

**Zimny strumień** gotowy do uruchomienia. Zawiera listę, przez którą przechodzi i emituje element co 3 sekundy

**Gorący strumień** zasilony przez zimny strumień topPlaysVideo

// 1. ZIMNY STRUMIEŃ: Gotowy plik wideo "Top 5 Klipów"

1 Usage

```
private val topPlaysVideo: Flow<String> = flow {  
    val clips = listOf("Klip #1: Stream 3!", "Klip #2: Stream 1!",  
        "Klip #3: Stream 5!", "Klip #4: Stream 12!")  
    for (clip in clips) {  
        emit(value = clip)  
        delay(timeMillis = 3000)  
    }  
}
```

// 2. GORĄCY STATEFLOW: "Kanał Twitch" retransmitujący wideo na żywo

1 Usage

```
val liveTopPlays: StateFlow<String> = topPlaysVideo  
    .stateIn(  
        scope = viewModelScope,  
        started = SharingStarted.WhileSubscribed(stopTimeoutMillis = 5000),  
        initialValue = "Retransmisja rozpocznie się za chwilę..."  
    )
```

# stateIn vs shareIn

**Zimny strumień** gotowy do uruchomienia. Zawiera listę, przez którą przechodzi i emituje element co 3 sekundy

**Gorący strumień** zasilony przez zimny strumień topPlaysVideo

Zamienia **zimny** Flow<String> w **gorący** StateFlow<T> z **ostatnią** znaną wartością.

// 1. ZIMNY STRUMIEŃ: Gotowy plik wideo "Top 5 Klipów"

1 Usage

```
private val topPlaysVideo: Flow<String> = flow {  
    val clips = listOf("Klip #1: Stream 3!", "Klip #2: Stream 1!",  
        "Klip #3: Stream 5!", "Klip #4: Stream 12!")  
    for (clip in clips) {  
        emit(value = clip)  
        delay(timeMillis = 3000)  
    }  
}
```

// 2. GORĄCY STATEFLOW: "Kanał Twitch" retransmitujący wideo na żywo

1 Usage

```
val liveTopPlays: StateFlow<String> = topPlaysVideo  
    .stateIn(  
        scope = viewModelScope,  
        started = SharingStarted.WhileSubscribed(stopTimeoutMillis = 5000),  
        initialValue = "Retransmisja rozpocznie się za chwilę..."  
    )
```

# stateIn vs shareIn

**Zimny strumień** gotowy do uruchomienia. Zawiera listę, przez którą przechodzi i emituje element co 3 sekundy

**Gorący strumień** zasilony przez zimny strumień topPlaysVideo

Zamienia **zimny** Flow<String> w **gorący** StateFlow<T> z **ostatnią znaną wartością**.

Uruchamia zbieranie w podanym **scope**. W tym przypadku strumień będzie **aktywny** tak długo jak **ViewModel**, i zostanie **automatycznie anulowany**, gdy **ViewModel** będzie niszczone.

// 1. ZIMNY STRUMIEŃ: Gotowy plik wideo "Top 5 Klipów"

1 Usage

```
private val topPlaysVideo: Flow<String> = flow {  
    val clips = listOf("Klip #1: Stream 3!", "Klip #2: Stream 1!",  
        "Klip #3: Stream 5!", "Klip #4: Stream 12!")  
    for (clip in clips) {  
        emit(value = clip)  
        delay(timeMillis = 3000)  
    }  
}
```

// 2. GORĄCY STATEFLOW: "Kanał Twitch" retransmitujący wideo na żywo

1 Usage

```
val liveTopPlays: StateFlow<String> = topPlaysVideo  
    .stateIn(  
        scope = viewModelScope,  
        started = SharingStarted.WhileSubscribed(stopTimeoutMillis = 5000),  
        initialValue = "Retransmisja rozpocznie się za chwilę..."  
    )
```

# stateIn vs shareIn

**Zimny strumień** gotowy do uruchomienia. Zawiera listę, przez którą przechodzi i emituje element co 3 sekundy

**Gorący strumień** zasilony przez zimny strumień topPlaysVideo

Zamienia **zimny** Flow<String> w **gorący** StateFlow<T> z **ostatnią znaną wartością**.

Uruchamia zbieranie w podanym **scope**. W tym przypadku strumień będzie **aktywny** tak długo jak **ViewModel**, i zostanie **automatycznie anulowany**, gdy **ViewModel** będzie niszczone.

Strategia uruchamiania:

- **Eagerly** – start od razu
- **Lazily** – start przy pierwszym subskrybencie
- **WhileSubscribed** – start przy **pierwszym subskrybencie** i zatrzymanie po **braku subskrybentów** przez stopTimeoutMillis

// 1. ZIMNY STRUMIEŃ: Gotowy plik wideo "Top 5 Klipów"

1 Usage

```
private val topPlaysVideo: Flow<String> = flow {  
    val clips = listOf("Klip #1: Stream 3!", "Klip #2: Stream 1!",  
        "Klip #3: Stream 5!", "Klip #4: Stream 12!")  
    for (clip in clips) {  
        emit(value = clip)  
        delay(timeMillis = 3000)  
    }  
}
```

// 2. GORĄCY STATEFLOW: "Kanał Twitch" retransmitujący wideo na żywo

1 Usage

```
val liveTopPlays: StateFlow<String> = topPlaysVideo  
    .stateIn(  
        scope = viewModelScope,  
        started = SharingStarted.WhileSubscribed(stopTimeoutMillis = 5000),  
        initialValue = "Retransmisja rozpocznie się za chwilę..."  
    )
```

# stateIn vs shareIn

**Zimny strumień** gotowy do uruchomienia. Zawiera listę, przez którą przechodzi i emituje element co 3 sekundy

**Gorący strumień** zasilony przez zimny strumień `topPlaysVideo`

Zamienia **zimny** `Flow<String>` w **gorący** `StateFlow<T>` z **ostatnią znaną wartością**.

Uruchamia zbieranie w podanym **scope**. W tym przypadku strumień będzie **aktywny** tak długo jak **ViewModel**, i zostanie **automatycznie anulowany**, gdy **ViewModel** będzie niszczone.

Strategia uruchamiania:

- **Eagerly** – start od razu
- **Lazily** – start przy pierwszym subskrybencie
- **WhileSubscribed** – start przy **pierwszym subskrybencie** i zatrzymanie po **braku subskrybentów** przez `stopTimeoutMillis`

// 1. ZIMNY STRUMIEŃ: Gotowy plik wideo "Top 5 Klipów"

1 Usage

```
private val topPlaysVideo: Flow<String> = flow {  
    val clips = listOf("Klip #1: Stream 3!", "Klip #2: Stream 1!",  
        "Klip #3: Stream 5!", "Klip #4: Stream 12!")  
    for (clip in clips) {  
        emit(value = clip)  
        delay(timeMillis = 3000)  
    }  
}
```

// 2. GORĄCY STATEFLOW: "Kanał Twitch" retransmitujący wideo na żywo

1 Usage

```
val liveTopPlays: StateFlow<String> = topPlaysVideo  
    .stateIn(  
        scope = viewModelScope,  
        started = SharingStarted.WhileSubscribed(stopTimeoutMillis = 5000),  
        initialValue = "Retransmisja rozpocznie się za chwilę..."  
    )
```

To jest wartość, którą nowi subskrybenci zobaczą, **zanim** zimny strumień wyemituje swój pierwszy element.

# stateIn vs shareIn

**Zimny strumień** gotowy do uruchomienia. Zawiera event, która chcemy uruchomić **pod pewnymi warunkami**.

// 1. ZIMNY STRUMIEŃ: Gotowa ścieżka dźwiękowa z alertami  
1 Usage

```
private val alertAudioTrack: Flow<String> = flow {  
    delay( timeMillis = 3000)  
    emit( value = "🔔 NOWY SUB!")  
    delay( timeMillis = 7000)  
    emit( value = "💎 NOWA DONACJA!")  
}
```

// 2. GORĄCY SHAREDFLOW: System powiadomień "na żywo"  
1 Usage

```
val liveAlerts: SharedFlow<String> = alertAudioTrack  
    .shareIn(  
        scope = viewModelScope,  
        started = SharingStarted.WhileSubscribed( stopTimeoutMillis = 5000)  
    )
```

# stateIn vs shareIn

**Zimny strumień** gotowy do uruchomienia. Zawiera event, która chcemy uruchomić **pod pewnymi warunkami**.

**Gorący strumień** zasilony przez zimny strumień alertAudioTrack

```
// 1. ZIMNY STRUMIEŃ: Gotowa ścieżka dźwiękowa z alertami  
1 Usage
```

```
private val alertAudioTrack: Flow<String> = flow {  
    delay( timeMillis = 3000)  
    emit( value = "🔔 NOWY SUB!")  
    delay( timeMillis = 7000)  
    emit( value = "💎 NOWA DONACJA!")  
}
```

```
// 2. GORĄCY SHAREDFLOW: System powiadomień "na żywo"  
1 Usage
```

```
val liveAlerts: SharedFlow<String> = alertAudioTrack  
    .shareIn(  
        scope = viewModelScope,  
        started = SharingStarted.WhileSubscribed( stopTimeoutMillis = 5000)  
    )
```



# stateIn vs shareIn

**Zimny strumień** gotowy do uruchomienia. Zawiera event, która chcemy uruchomić **pod pewnymi warunkami**.

**Gorący strumień** zasilony przez zimny strumień `alertAudioTrack`

Pozwala nam **współdzielić** jeden strumień danych pomiędzy wieloma subskrybentami. Wszyscy **aktywni obserwatorzy** (np. różne części UI) otrzymają powiadomienie **w tym samym momencie**. Co najważniejsze, `SharedFlow` domyślnie **nie odtwarza starych zdarzeń**

// 1. ZIMNY STRUMIEŃ: Gotowa ścieżka dźwiękowa z alertami  
1 Usage

```
private val alertAudioTrack: Flow<String> = flow {  
    delay( timeMillis = 3000)  
    emit( value = "🔔 NOWY SUB!")  
    delay( timeMillis = 7000)  
    emit( value = "💎 NOWA DONACJA!")  
}
```

// 2. GORĄCY SHAREDFLOW: System powiadomień "na żywo"  
1 Usage

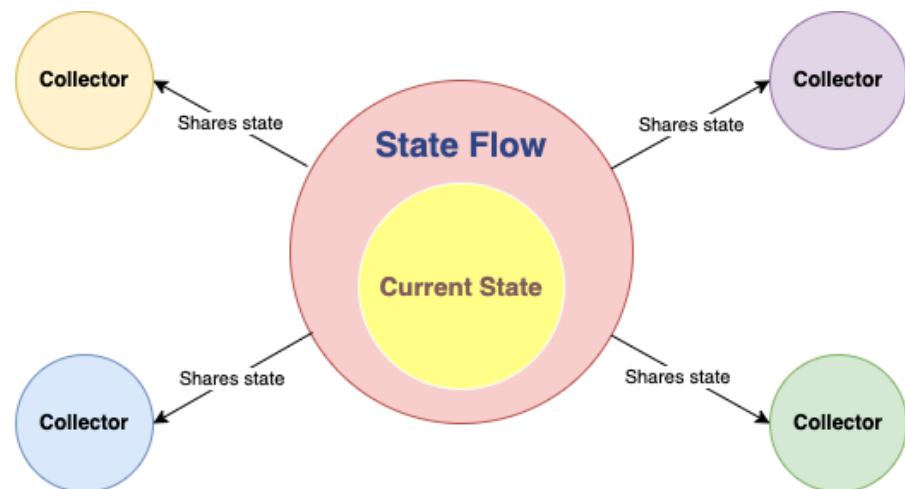
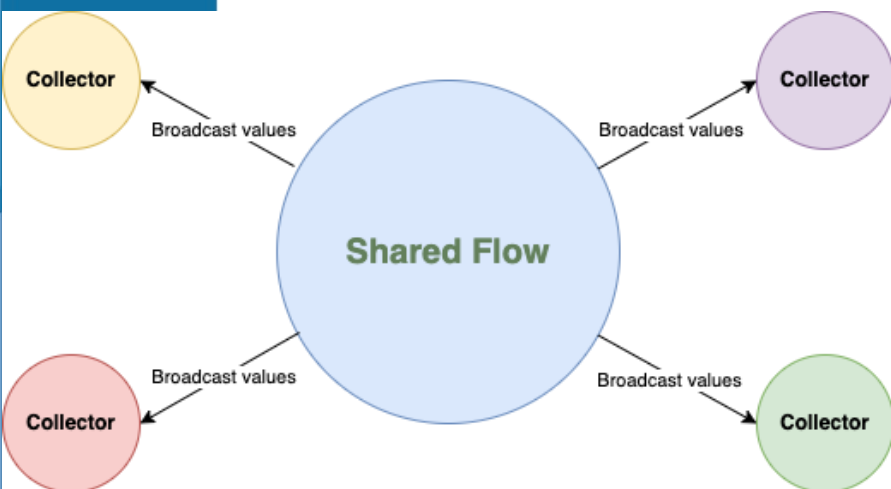
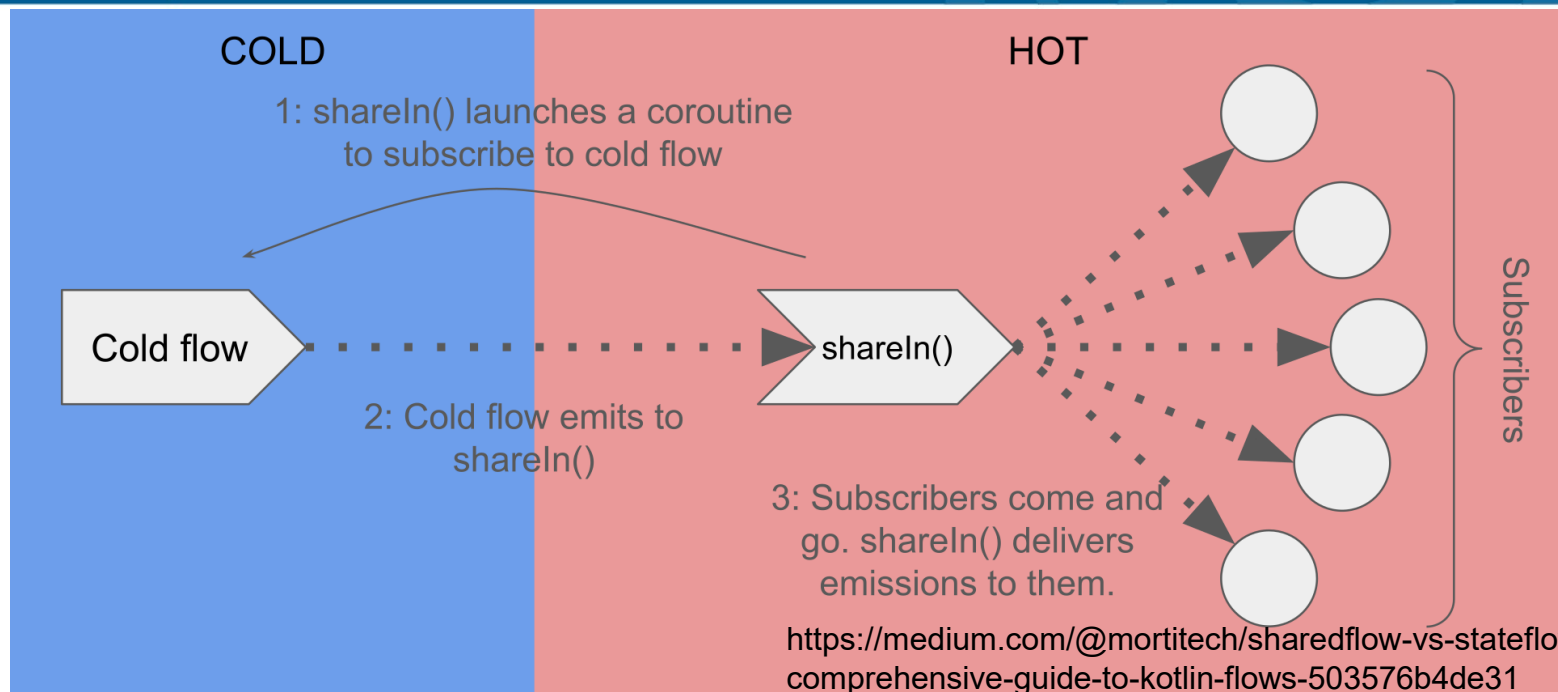
```
val liveAlerts: SharedFlow<String> = alertAudioTrack  
    .shareIn(  
        scope = viewModelScope,  
        started = SharingStarted.WhileSubscribed( stopTimeoutMillis = 5000)  
    )
```



# stateIn vs shareIn

Cecha	shareIn	stateIn
Główne Zastosowanie	Wysyłanie <b>zdarzeń</b> jednorazowych (eventów), które nie powinny być odtwarzane (np. Snackbar, nawigacja).	Reprezentowanie i udostępnianie <b>stanu</b> UI, który zawsze ma aktualną wartość (np. dane profilu, zawartość koszyka).
Posiadanie Wartości	Nie musi mieć wartości w momencie subskrypcji. Emituje wartości, gdy pojawią się w źródłowym strumieniu.	<b>Zawsze</b> ma aktualną wartość, którą można w dowolnym momencie odczytać.
Wartość Początkowa	Nie wymaga wartości początkowej.	<b>Wymaga</b> podania wartości początkowej (initialValue).
Odtwarzanie dla Nowych Subskrybentów	Domyślnie <b>nie odtwarza</b> starych wartości. Nowy subskrybent czeka na nowe emisje.	<b>Zawsze odtwarza</b> ostatnio zapisaną wartość dla nowego subskrybenta.
Typ Zwracany	SharedFlow<T>	StateFlow<T>
Analogia	<b>Dzwonek do drzwi</b> lub <b>alert na Twitchu</b> (zdarzenie jednorazowe).	<b>Telebim giełdowy</b> lub <b>główny ekran gry na Twitchu</b> (zawsze widoczny stan).

# stateIn vs shareIn



Stworzenie **jednego** stanu, który zależy od **wielu niezależnych źródeł danych**.

Chcemy wyświetlić dane użytkownika (`Flow<User>`) oraz jego powiadomienia (`Flow<List<Notification>>`). Funkcja `combine` pozwala **połączyć** te dwa strumienie w jeden `Flow<UiState>`.

```
val user: Flow<User> = userRepository.getUserStream()
val settings: Flow<Settings> = settingsRepository.getSettingsStream()

val uiState: StateFlow<UiState> = combine(user, settings) { currentUser, currentSettings ->
    UiState(userName = currentUser.name, notificationsEnabled = currentSettings.notifications)
}.stateIn(...)
```

Za każdym razem, gdy `user` lub `settings` się zmieniają, blok `combine` wyemituje nowy, **połączony stan**.

Określa na którym dispatcherze ma być wykonana operacja. **Zmienia dispatcher** korutyn dla **upstreamu** – wszystkiego, co jest **przed** nim w łańcuchu. **Nie zmienia** kontekstu **kolektora** i wprowadza **bufor** między segmentami.

Utworzenie **zimnego strumienia** w repozytorium. Rozpoczyna na Dispatchers.Main

```
data class User(val id: Int, val name: String)
```

2 Usages

```
class UserRepository {
```

```
    // Ta funkcja zwraca ZIMNY Flow.
```

```
    // Sama w sobie nie przełącza wątków.
```

1 Usage

```
    fun getUsersStream(): Flow<List<User>> = flow {  
        delay( timeMillis = 1500)  
        emit( value = listOf(  
            User( id = 1, name = "Anna"),  
            User( id = 2, name = "Piotr"),  
            User( id = 3, name = "Zofia"))) )  
    }
```

```
val uppercasedUserNames: StateFlow<List<String>> =  
    repository.getUsersStream()  
        .flowOn( context = Dispatchers.IO )  
        .map { users ->  
            users.map { it.name.uppercase() }  
        }  
        .stateIn(  
            scope = viewModelScope,  
            started = SharingStarted.WhileSubscribed( stopTimeoutMillis = 5000 ),  
            initialValue = emptyList()  
        )
```

Określa na którym dispatcherze ma być wykonana operacja. **Zmienia dispatcher** korutyn dla **upstreamu** – wszystkiego, co jest **przed** nim w łańcuchu. **Nie zmienia** kontekstu **kolektora** i wprowadza **bufor** między segmentami.

Utworzenie **zimnego strumienia** w repozytorium. Rozpoczyna na Dispatchers.Main

**Zimny strumień** zwrócony z repozytorium.

```
data class User(val id: Int, val name: String)
2 Usages
class UserRepository {
    // Ta funkcja zwraca ZIMNY Flow.
    // Sama w sobie nie przełącza wątków.
    1 Usage
    fun getUsersStream(): Flow<List<User>> = flow {
        delay( timeMillis = 1500)
        emit( value = listOf(
            User( id = 1, name = "Anna"),
            User( id = 2, name = "Piotr"),
            User( id = 3, name = "Zofia")))
    }
}
```

```
val uppercasedUserNames: StateFlow<List<String>> =
    repository.getUsersStream()
        .flowOn( context = Dispatchers.IO)
        .map { users ->
            users.map { it.name.uppercase() }
        }
        .stateIn(
            scope = viewModelScope,
            started = SharingStarted.WhileSubscribed( stopTimeoutMillis = 5000),
            initialValue = emptyList()
        )
)
```

Określa na którym dispatcherze ma być wykonana operacja. **Zmienia dispatcher** korutyn dla **upstreamu** – wszystkiego, co jest **przed** nim w łańcuchu. **Nie zmienia** kontekstu **kolektora** i wprowadza **bufor** między segmentami.

Utworzenie **zimnego strumienia** w repozytorium. Rozpoczyna na Dispatchers.Main

**Zimny strumień** zwrócony z repozytorium.

Dzięki `flowOn(Dispatchers.IO)`, kod **wewnątrz flow** w `UserRepository` został wykonany **na wątku w tle z puli IO. (Upstream)**

```
data class User(val id: Int, val name: String)
```

2 Usages

```
class UserRepository {
```

```
    // Ta funkcja zwraca ZIMNY Flow.
```

```
    // Sama w sobie nie przełącza wątków.
```

1 Usage

```
    fun getUsersStream(): Flow<List<User>> = flow {
```

```
        delay( timeMillis = 1500)
```

```
        emit( value = listOf(
```

```
            User( id = 1, name = "Anna"),
```

```
            User( id = 2, name = "Piotr"),
```

```
            User( id = 3, name = "Zofia")))
```

```
    }
```

```
}
```

```
val uppercaseUserNames: StateFlow<List<String>> =
```

```
    repository.getUsersStream()
```

```
    .flowOn( context = Dispatchers.IO)
```

```
    .map { users ->
```

```
        users.map { it.name.uppercase() }
```

```
    }
```

```
    .stateIn(
```

```
        scope = viewModelScope,
```

```
        started = SharingStarted.WhileSubscribed( stopTimeoutMillis = 5000),
```

```
        initialValue = emptyList()
```

```
)
```

Określa na którym dispatcherze ma być wykonana operacja. **Zmienia dispatcher** korutyn dla **upstreamu** – wszystkiego, co jest **przed** nim w łańcuchu. **Nie zmienia** kontekstu **kolektora** i wprowadza **bufor** między segmentami.

Utworzenie **zimnego strumienia** w repozytorium. Rozpoczyna na Dispatchers.Main

**Zimny strumień** zwrócony z repozytorium.

Dzięki `flowOn(Dispatchers.IO)`, kod **wewnątrz flow** w `UserRepository` został wykonany **na wątku w tle z puli IO. (Upstream)**

`.map` znajduje się **poniżej** `flowOn` w łańcuchu wywołań. Oznacza to, że jest on częścią **downstream** i wykonuje się na wątku Main.

```
data class User(val id: Int, val name: String)
2 Usages
class UserRepository {
    // Ta funkcja zwraca ZIMNY Flow.
    // Sama w sobie nie przełącza wątków.
    1 Usage
    fun getUsersStream(): Flow<List<User>> = flow {
        delay( timeMillis = 1500)
        emit( value = listOf(
            User( id = 1, name = "Anna"),
            User( id = 2, name = "Piotr"),
            User( id = 3, name = "Zofia")))
    }
}

val uppercasedUserNames: StateFlow<List<String>> =
    repository.getUsersStream()
    .flowOn( context = Dispatchers.IO)
    .map { users ->
        users.map { it.name.uppercase() }
    }
    .stateIn(
        scope = viewModelScope,
        started = SharingStarted.WhileSubscribed( stopTimeoutMillis = 5000),
        initialValue = emptyList()
    )
)
```

# flowOn vs withContext

Aspekt	flowOn	withContext
Cel	Zmiana kontekstu <b>upstreamu</b> w Flow	Zmiana kontekstu <b>bieżącej</b> korutyny na czas bloku
Zakres działania	Tylko funkcje <b>przed</b> flowOn	Całe wnętrze bloku withContext { ... }
Wpływ na operatorów	Upstream przeniesiony, downstream bez zmian	Wszystko w bloku przeniesione
Buforowanie	Tak, dodaje bufor między segmentami	Nie, brak dodatkowego bufora
Typ użycia	Operator w łańcuchu Flow	Każda suspend fun lub sekcja kodu
Wielokrotność	Można łańcuchować kilka flowOn	Można zagnieżdżać, ale rzadko potrzebne