Uniwersytet Wrocławski

Rafał Lewandków

PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 2

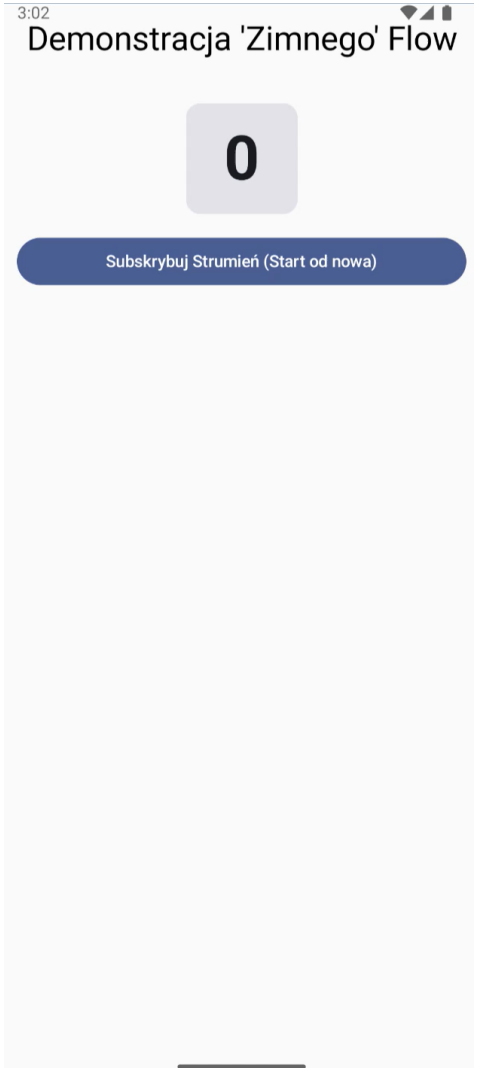
WYKŁAD 5

Reaktywne Zarządzanie Stanem:

- Flow
- StateFlow
- SharedFlow

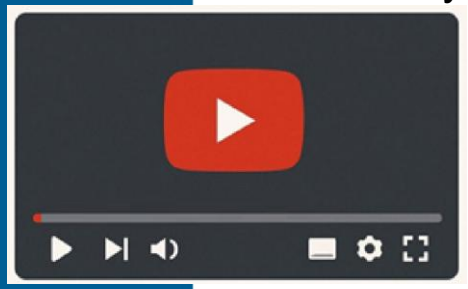
Flow

Do tej pory nasze funkcje suspend zwracały **pojedynczą wartość** (np. String lub List). A co, jeśli chcemy, aby funkcja zwracała **sekwencję wartości** w czasie? Do tego właśnie służy **Flow** – **asynchroniczny strumień danych**.



Flow

Do tej pory nasze funkcje suspend zwracały **pojedynczą wartość** (np. String lub List). A co, jeśli chcemy, aby funkcja zwracała **sekwencję wartości** w czasie? Do tego właśnie służy **Flow** – **asynchroniczny strumień danych**.

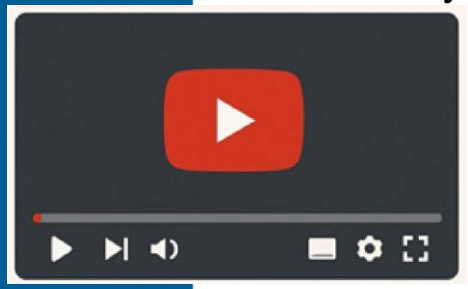


To jest film na kanale YouTube o tytule numberStream i istnieje jako gotowy do odtworzenia materiał. Samo jego istnienie **nic nie robi** – nikt go jeszcze nie ogląda. Jest ***"zimny".***

```
fun numberStream(): Flow<Int> = flow {
    println("Flow: Strumień wystartował!") // To wydrukuje się przy każdej
    for (i in 1..3) {
        delay(1000)
        emit(i) // 'emit()' wysyła wartość do strumienia
    }
}

// Uruchomienie wewnątrz korutyny
scope.launch {
    println("Zaczynam słuchać strumienia...")
    numberStream().collect { number ->
        println("Odebrano liczbę: $number")
    }
    println("Strumień zakończony.")
}
```

Flow

Do tej pory nasze funkcje suspend zwracały **pojedynczą wartość** (np. String lub List). A co, jeśli chcemy, aby funkcja zwracała **sekwencję wartości** w czasie? Do tego właśnie służy **Flow** – **asynchroniczny strumień danych**.



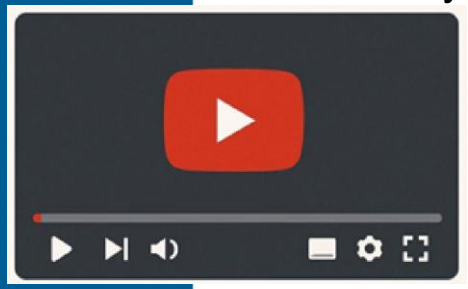
To jest film na kanale YouTube o tytule numberStream i istnieje jako gotowy do odtworzenia materiał. Samo jego istnienie **nic nie robi** – nikt go jeszcze nie ogląda. Jest ***"zimny".***

To jest treść filmu

```
fun numberStream(): Flow<Int> = flow {
    println("Flow: Strumień wystartował!") // To wydrukuje się przy każdej
    for (i in 1..3) {
        delay(1000)
        emit(i) // 'emit()' wysyła wartość do strumienia
    }
}

// Uruchomienie wewnątrz korutyny
scope.launch {
    println("Zaczynam słuchać strumienia...")
    numberStream().collect { number ->
        println("Odebrano liczbę: $number")
    }
    println("Strumień zakończony.")
}
```

Flow

Do tej pory nasze funkcje suspend zwracały **pojedynczą wartość** (np. String lub List). A co, jeśli chcemy, aby funkcja zwracała **sekwencję wartości** w czasie? Do tego właśnie służy **Flow** – **asynchroniczny strumień danych**.



To jest film na kanale YouTube o tytule numberStream i istnieje jako gotowy do odtworzenia materiał. Samo jego istnienie **nic nie robi** – nikt go jeszcze nie ogląda. Jest ***"zimny".***

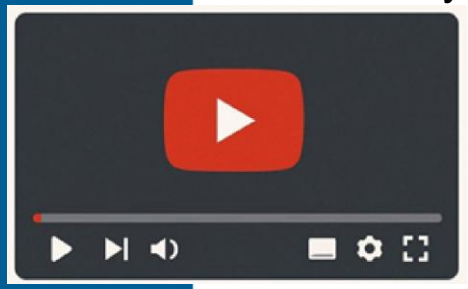
To jest treść filmu

Co sekundę na ekranie pojawia się nowa liczba. emit to akt ***"wyemitowania"*** klatki filmu.

```
fun numberStream(): Flow<Int> = flow {
    println("Flow: Strumień wystartował!") // To wydrukuje się przy każdej
    for (i in 1..3) {
        delay(1000)
        emit(i) // 'emit()' wysyła wartość do strumienia
    }
}

// Uruchomienie wewnątrz korutyny
scope.launch {
    println("Zaczynam słuchać strumienia...")
    numberStream().collect { number ->
        println("Odebrano liczbę: $number")
    }
    println("Strumień zakończony.")
}
```

Flow

Do tej pory nasze funkcje suspend zwracały **pojedynczą wartość** (np. String lub List). A co, jeśli chcemy, aby funkcja zwracała **sekwencję wartości** w czasie? Do tego właśnie służy **Flow** – **asynchroniczny strumień danych**.



To jest film na kanale YouTube o tytule numberStream i istnieje jako gotowy do odtworzenia materiał. Samo jego istnienie **nic nie robi** – nikt go jeszcze nie ogląda. Jest ***"zimny".***

To jest treść filmu

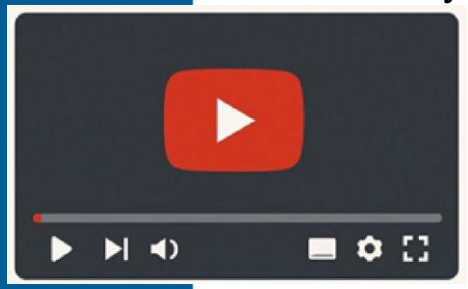
Co sekundę na ekranie pojawia się nowa liczba. emit to akt ***"wyemitowania"*** klatki filmu.

Załadowanie linku z interesującym nas filmem.

```
fun numberStream(): Flow<Int> = flow {
    println("Flow: Strumień wystartował!") // To wydrukuje się przy każdej
    for (i in 1..3) {
        delay(1000)
        emit(i) // 'emit()' wysyła wartość do strumienia
    }
}

// Uruchomienie wewnątrz korutyny
scope.launch {
    println("Zaczynam słuchać strumienia...")
    numberStream().collect { number ->
        println("Odebrano liczbę: $number")
    }
    println("Strumień zakończony.")
}
```

Flow

Do tej pory nasze funkcje suspend zwracały **pojedynczą wartość** (np. String lub List). A co, jeśli chcemy, aby funkcja zwracała **sekwencję wartości** w czasie? Do tego właśnie służy **Flow** – **asynchroniczny strumień danych**.



To jest film na kanale YouTube o tytule numberStream i istnieje jako gotowy do odtworzenia materiał. Samo jego istnienie **nic nie robi** – nikt go jeszcze nie ogląda. Jest ***"zimny".***

To jest treść filmu

Co sekundę na ekranie pojawia się nowa liczba. emit to akt ***"wyemitowania"*** klatki filmu.

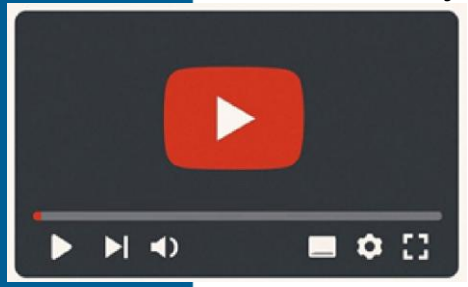
Załadowanie linku z interesującym nas filmem.

To jest moment, w którym widz wciska przycisk ***"Play„.***

```
fun numberStream(): Flow<Int> = flow {
    println("Flow: Strumień wystartował!") // To wydrukuje się przy każdej
    for (i in 1..3) {
        delay(1000)
        emit(i) // 'emit()' wysyła wartość do strumienia
    }
}

// Uruchomienie wewnątrz korutyny
scope.launch {
    println("Zaczynam słuchać strumienia...")
    numberStream().collect { number ->
        println("Odebrano liczbę: $number")
    }
    println("Strumień zakończony.")
}
```

Flow

Do tej pory nasze funkcje suspend zwracały **pojedynczą wartość** (np. String lub List). A co, jeśli chcemy, aby funkcja zwracała **sekwencję wartości** w czasie? Do tego właśnie służy **Flow** – **asynchroniczny strumień danych**.



To jest film na kanale YouTube o tytule numberStream i istnieje jako gotowy do odtworzenia materiał. Samo jego istnienie **nic nie robi** – nikt go jeszcze nie ogląda. Jest ***"zimny".***

To jest treść filmu

Co sekundę na ekranie pojawia się nowa liczba. emit to akt ***"wyemitowania"*** klatki filmu.

```
fun numberStream(): Flow<Int> = flow {
    println("Flow: Strumień wystartował!") // To wydrukuje się przy każdej
    for (i in 1..3) {
        delay(1000)
        emit(i) // 'emit()' wysyła wartość do strumienia
    }
}

// Uruchomienie wewnątrz korutyny
scope.launch {
    println("Zaczynam słuchać strumienia...")
    numberStream().collect { number ->
        println("Odebrano liczbę: $number")
    }
    println("Strumień zakończony.")
```

Załadowanie linku z interesującym nas filmem.

To jest moment, w którym widz wciska przycisk ***"Play„.***

Gdyby inny "*widz*" (inna korutyna) również wywołał numberStream().collect(), obejrzałby **ten sam film od początku**, we własnym tempie. To jest esencja "***zimnego***" strumienia Flow.

Flow

Do tej pory nasze funkcje suspend zwracały **pojedynczą wartość** (np. String lub List). A co, jeśli chcemy, aby funkcja zwracała **sekwencję wartości** w czasie? Do tego właśnie służy **Flow** – **asynchroniczny strumień danych**.

```
// Definicja "zimnego" strumienia - to jest tylko przepis.
val coldStream: Flow<String> = flow {
    println("=> PRZEPIS: Rozpoczynam przygotowanie...")
    delay(100)
    emit("Składnik A")
    delay(100)
    emit("Składnik B")
    println("=> PRZEPIS: Zakończono.")
}

fun main() = runBlocking {
    println("--- PIECZENIE #1 ---")
    // Pierwsza osoba "piecze ciasto" (subskrybuje strumień)
    coldStream.collect { skladnik ->
        println("Kucharz 1 otrzymał: $skladnik")
    }

    println("\n--- PIECZENIE #2 ---")
    // Druga osoba "piecze ciasto" od nowa
    coldStream.collect { skladnik ->
        println("Kucharz 2 otrzymał: $skladnik")
    }
}
```

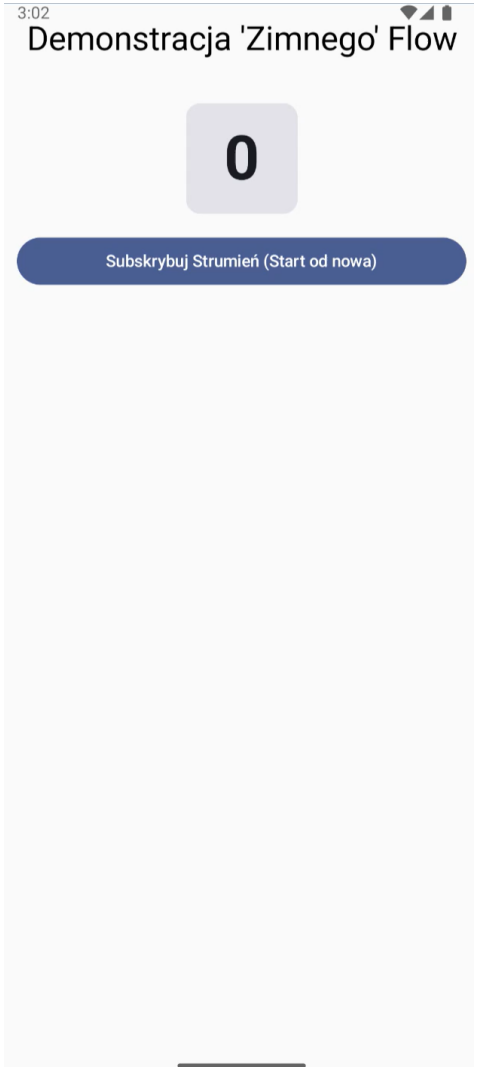
Flow

Do tej pory nasze funkcje suspend zwracały **pojedynczą wartość** (np. String lub List). A co, jeśli chcemy, aby funkcja zwracała **sekwencję wartości** w czasie? Do tego właśnie służy **Flow** – **asynchroniczny strumień danych**.

```
// Definicja "zimnego" strumienia - to jest tylko przepis.
val coldStream: Flow<String> = flow {
    println("=> PRZEPIS: Rozpoczynam przygotowanie...")
    delay(100)
    emit("Składnik A")
    delay(100)
    emit("Składnik B")
    println("=> PRZEPIS: Zakończono.")
}

fun main() = runBlocking {
    println("--- PIECZENIE #1 ---")
    // Pierwsza osoba "piecze ciasto" (subskrybuje strumień)
    coldStream.collect { skladnik ->
        println("Kucharz 1 otrzymał: $skladnik")
    }

    println("\n--- PIECZENIE #2 ---")
    // Druga osoba "piecze ciasto" od nowa
    coldStream.collect { skladnik ->
        println("Kucharz 2 otrzymał: $skladnik")
    }
}
```

```
--- PIECZENIE #1 ---
=> PRZEPIS: Rozpoczynam przygotowanie...
Kucharz 1 otrzymał: Składnik A
Kucharz 1 otrzymał: Składnik B
=> PRZEPIS: Zakończono.

--- PIECZENIE #2 ---
=> PRZEPIS: Rozpoczynam przygotowanie...
Kucharz 2 otrzymał: Składnik A
Kucharz 2 otrzymał: Składnik B
=> PRZEPIS: Zakończono.
```

Flow




```
@Composable
fun FlowDemoScreen() {
    val scope = rememberCoroutineScope()
    val logs = remember { mutableStateListOf<String>() }
    var latestValue by remember { mutableStateOf( value = 0) }
    val listState = rememberLazyListState()

    LaunchedEffect( key1 = logs.size) {
        if (logs.isNotEmpty()) {
            listState.animateScrollToItem( index = logs.size - 1)
        }
    }

    Column(...) {
        Text(...)
        Spacer(Modifier.height( height = 20.dp))

        Card(modifier = Modifier.padding( all = 16.dp)) {...}

        Button(
            onClick = {
                scope.launch {
                    logs.add("--- NOWA SUBSKRYPCJA ---")
                    numberStream(logs).collect { number ->
                        latestValue = number
                    }
                }
            },
            modifier = Modifier.fillMaxWidth()
        ) {...}
```

Uniwersytet Wrocławski

Flow

Obiekt **stanu**, który pozwala programowo kontrolować pozycję przewijania LazyColumn

```
@Composable
fun FlowDemoScreen() {
    val scope = rememberCoroutineScope()
    val logs = remember { mutableStateListOf<String>() }
    var latestValue by remember { mutableStateOf( value = 0) }
    val listState = rememberLazyListState()

    LaunchedEffect( key1 = logs.size) {
        if (logs.isNotEmpty()) {
            listState.animateScrollToItem( index = logs.size - 1)
        }
    }

    Column(...) {
        Text(...)
        Spacer(Modifier.height( height = 20.dp))

        Card(modifier = Modifier.padding( all = 16.dp)) {...}

        Button(
            onClick = {
                scope.launch {
                    logs.add("--- NOWA SUBSKRYPCJA ---")
                    numberStream(logs).collect { number ->
                        latestValue = number
                    }
                }
            },
            modifier = Modifier.fillMaxWidth()
        ) {...}
```

Flow

Obiekt **stanu**, który pozwala programowo kontrolować pozycję przewijania LazyColumn

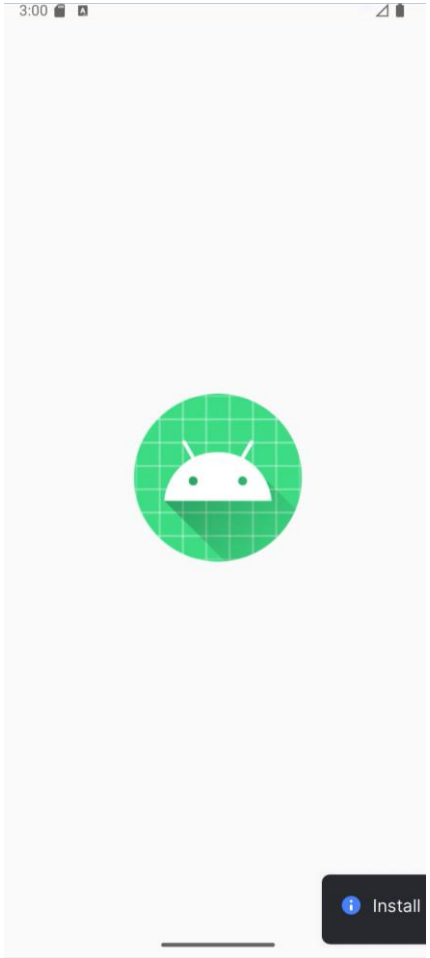
LaunchedEffect służy do uruchamiania **efektów ubocznych** (takich jak animacje, wywołania sieciowe, itp.) w odpowiedzi na **zmiany stanu** lub pojawienie się komponentu na ekranie.
W tym konkretnym przypadku rozwiązuje on następujący problem: gdy do listy logs dodawane są nowe wpisy, mogą one pojawić się na dole, poza widocznym obszarem LazyColumn. Użytkownik musiałby ręcznie przewijać listę, aby zobaczyć najnowsze logi.

```
@Composable
fun FlowDemoScreen() {
    val scope = rememberCoroutineScope()
    val logs = remember { mutableStateListOf<String>() }
    var latestValue by remember { mutableStateOf( value = 0) }
    val listState = rememberLazyListState()

    LaunchedEffect( key1 = logs.size) {
        if (logs.isNotEmpty()) {
            listState.animateScrollToItem( index = logs.size - 1)
        }
    }

    Column(...) {
        Text(...)
        Spacer(Modifier.height( height = 20.dp))

        Card(modifier = Modifier.padding( all = 16.dp)) {...}

        Button(
            onClick = {
                scope.launch {
                    logs.add("--- NOWA SUBSKRYPCJA ---")
                    numberStream(logs).collect { number ->
                        latestValue = number
                    }
                }
            },
            modifier = Modifier.fillMaxWidth()
        ) {...}
```

StateFlow

Flow jest świetny do **jednorazowych operacji**, które mają zwrócić wiele wyników (np. pobieranie pliku i emitowanie postępu w procentach). Ale **nie nadaje się** do przechowywania **stanu UI**, ponieważ każdy nowy obserwator (np. po obrocie ekranu) **uruchamiałby go od nowa**.

StateFlow to specjalny, ***"gorący"*** strumień, zaprojektowany do przechowywania i udostępniania stanu.



StateFlow

W przeciwieństwie do "*zimnego*" Flow, który jest tylko przepisem, "*gorący*" strumień musi być **aktywnie zarządzany**.

Licznik widzów na pulpicie streamera. **Zawsze ma jakąś wartość** (zaczyna od 0)

```
class TwitchStreamer {
    private val _viewerCount = MutableStateFlow(0)
    val viewerCount: StateFlow<Int> = _viewerCount.asStateFlow()

    init {
        GlobalScope.launch {
            while (true) {
                _viewerCount.value = Random.nextInt(100, 1000)
                delay(2000)
            }
        }
    }
}
```

StateFlow

W przeciwieństwie do "*zimnego*" Flow, który jest tylko przepisem, "*gorący*" strumień musi być **aktywnie zarządzany**.

konwertuje modyfikowalny MutableStateFlow na jego publiczną, niemodyfikowalną (tylko do odczytu) wersję StateFlow.

Jest to mechanizm ochronny, który uniemożliwia modyfikację stanu z zewnątrz klasy, która jest jego właścicielem (zazwyczaj z ViewModelu).

Licznik widzów na pulpicie streamera. **Zawsze ma jakąś wartość** (zaczyna od 0)

```
class TwitchStreamer {
    private val _viewerCount = MutableStateFlow(0)
    val viewerCount: StateFlow<Int> = _viewerCount.asStateFlow()

    init {
        GlobalScope.launch {
            while (true) {
                _viewerCount.value = Random.nextInt(100, 1000)
                delay(2000)
            }
        }
}
```

wzorzec z _viewerCount i viewerCount zapewnia **niemodyfikowalność** (*read-only*) stanu dla warstwy UI, co jest niemożliwe do osiągnięcia przy użyciu private set na MutableStateFlow.

private set chroni tylko przed przypisaniem nowego obiektu MutableStateFlow do zmiennej viewerCount z zewnątrz. Jednak UI (np. Composable) wciąż otrzymuje obiekt typu MutableStateFlow. Oznacza to, że wciąż może nadpisać wartość.

StateFlow

W przeciwieństwie do "*zimnego*" Flow, który jest tylko przepisem, "*gorący*" strumień musi być **aktywnie zarządzany**.

konwertuje modyfikowalny MutableStateFlow na jego publiczną, niemodyfikowalną (tylko do odczytu) wersję StateFlow.

Jest to mechanizm ochronny, który uniemożliwia modyfikację stanu z zewnątrz klasy, która jest jego właścicielem (zazwyczaj z ViewModelu).

Licznik widzów na pulpicie streamera. **Zawsze ma jakąś wartość** (zaczyna od 0)

```
class TwitchStreamer {
    private val _viewerCount = MutableStateFlow(0)
    val viewerCount: StateFlow<Int> = _viewerCount.asStateFlow()

    init {
        GlobalScope.launch {
            while (true) {
                _viewerCount.value = Random.nextInt(100, 1000)
                delay(2000)
            }
        }
    }
```

wzorzec z _viewerCount i viewerCount zapewnia **niemodyfikowalność** (*read-only*) stanu dla warstwy UI, co jest niemożliwe do osiągnięcia przy użyciu private set na MutableStateFlow.

private set chroni tylko przed przypisaniem nowego obiektu MutableStateFlow do zmiennej viewerCount z zewnątrz. Jednak UI (np. Composable) wciąż otrzymuje obiekt typu MutableStateFlow. Oznacza to, że wciąż może nadpisać wartość.

Gdy tylko tworzony jest obiekt TwitchStreamer, natychmiast uruchamia w tle korutynę, która w pętli symuluje zmiany liczby widzów. Transmisja ruszyła i jest "*na żywo*".

GlobalScope jest używany tylko dla uproszczenia tego przykładu.

StateFlow

Streamer rozpoczyna transmisję. Jego licznik widzów już zaczyna się zmieniać

```
fun main() = runBlocking {
    println("▶️ Streamer rozpoczyna transmisję...")
    val streamer = TwitchStreamer()

    println("...nikt jeszcze nie ogląda, ale licznik widzów już się zmienia w tle.
    delay(5000)

    launch {
        println("😊 Widz 1 dołączył!")
        streamer.viewerCount.collect { count ->
            println("[Widz 1] Patrzy na licznik: $count widzów")
        }
    }

    delay(4000)

    launch {
        println("😃 Widz 2 dołączył!")
        streamer.viewerCount.collect { count ->
            println("[Widz 2] Patrzy na licznik: $count widzów")
        }
    }
}
```

StateFlow

Streamer rozpoczyna transmisję. Jego licznik widzów już zaczyna się zmieniać

Transmisja trwa, mimo że nikt jej nie subskrybuje.

```
fun main() = runBlocking {
    println("▶️ Streamer rozpoczyna transmisję...")
    val streamer = TwitchStreamer()

    println("...nikt jeszcze nie ogląda, ale licznik widzów już się zmienia w tle.
    delay(5000)

    launch {
        println("😐 Widz 1 dołączył!")
        streamer.viewerCount.collect { count ->
            println("[Widz 1] Patrzy na licznik: $count widzów")
        }
    }

    delay(4000)

    launch {
        println("😃 Widz 2 dołączył!")
        streamer.viewerCount.collect { count ->
            println("[Widz 2] Patrzy na licznik: $count widzów")
        }
    }
}
```

StateFlow


```
fun main() = runBlocking {
    println("▶️ Streamer rozpoczyna transmisję...")
    val streamer = TwitchStreamer()

    println("...nikt jeszcze nie ogląda, ale licznik widzów już się zmienia w tle.
    delay(5000)

    launch {
        println("😊 Widz 1 dołączył!")
        streamer.viewerCount.collect { count ->
            println("[Widz 1] Patrzy na licznik: $count widzów")
        }
    }

    delay(4000)

    launch {
        println("😃 Widz 2 dołączył!")
        streamer.viewerCount.collect { count ->
            println("[Widz 2] Patrzy na licznik: $count widzów")
        }
    }
}
```

Streamer rozpoczyna transmisję. Jego licznik widzów już zaczyna się zmieniać

Transmisja trwa, mimo że nikt jej nie subskrybuje.

Pierwszy widz wchodzi na kanał. Nie widzi licznika od 0. Natychmiast dostaje **aktualną wartość licznika**, np. 458.

StateFlow


```
fun main() = runBlocking {
    println("▶️ Streamer rozpoczyna transmisję...")
    val streamer = TwitchStreamer()

    println("...nikt jeszcze nie ogląda, ale licznik widzów już się zmienia w tle.")
    delay(5000)

    launch {
        println("😊 Widz 1 dołączył!")
        streamer.viewerCount.collect { count ->
            println("[Widz 1] Patrzy na licznik: $count widzów")
        }
    }

    delay(4000)

    launch {
        println("😃 Widz 2 dołączył!")
        streamer.viewerCount.collect { count ->
            println("[Widz 2] Patrzy na licznik: $count widzów")
        }
    }
}
```

Streamer rozpoczyna transmisję. Jego licznik widzów już zaczyna się zmieniać

Transmisja trwa, mimo że nikt jej nie subskrybuje.

Pierwszy widz wchodzi na kanał. Nie widzi licznika od 0. Natychmiast dostaje **aktualną wartość licznika**, np. 458.

Drugi widz wchodzi na kanał. On również natychmiast dostaje aktualną wartość, np. 721, i od tego momentu **obaj widzowie widzą te same, synchroniczne aktualizacje**.

Uniwersytet Wrocławski

StateFlow

StateFlow: jest "***gorący***", **zawsze** ma **aktualny stan**, który natychmiast **udostępnia nowym subskrybentom**.

```
fun main() = runBlocking {
    println("▶️ Streamer rozpoczyna transmisję...")
    val streamer = TwitchStreamer()

    println("...nikt jeszcze nie ogląda, ale licznik widzów już się zmienia w tle.
    delay(5000)

    launch {
        println("😊 Widz 1 dołączył!")
        streamer.viewerCount.collect { count ->
            println("[Widz 1] Patrzy na licznik: $count widzów")
        }
    }

    delay(4000)

    launch {
        println("😃 Widz 2 dołączył!")
        streamer.viewerCount.collect { count ->
            println("[Widz 2] Patrzy na licznik: $count widzów")
        }
    }
}
```

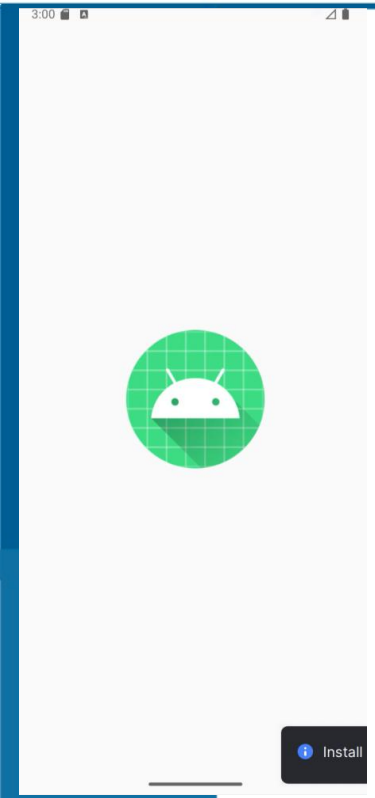
Streamer rozpoczyna transmisję. Jego licznik widzów już zaczyna się zmieniać

Transmisja trwa, mimo że nikt jej nie subskrybuje.

Pierwszy widz wchodzi na kanał. Nie widzi licznika od 0. Natychmiast dostaje **aktualną wartość licznika**, np. 458.

Drugi widz wchodzi na kanał. On również natychmiast dostaje aktualną wartość, np. 721, i od tego momentu **obaj widzowie widzą te same, synchroniczne aktualizacje**.

StateFlow




```
class HotStreamViewModel : ViewModel() {
    // Prywatny, modyfikowalny StateFlow - licznik, który widzi tylko streamer.
    2 Usages
    private val _numberStream = MutableStateFlow( value = 0)
    // Publiczny, niemodyfikowalny StateFlow - to widzi publiczność.
    3 Usages
    val numberStream: StateFlow<Int> = _numberStream.asStateFlow()

    init {
        // Streamer rozpoczyna "transmisję na żywo" w momencie utworzenia ViewModelu.
        // Strumień jest "gorący" - działa niezależnie od liczby widzów (subskrybentów).
        viewModelScope.launch {
            var count = 0
            while (true) {
                delay( timeMillis = 1000)
                count++
                _numberStream.value = count
            }
        }
    }
}
1 Usage
@Composable
fun StateFlowDemoScreen(viewModel: HotStreamViewModel = viewModel()) {
    val liveValue by viewModel.numberStream.collectAsStateWithLifecycle()
```

viewModelScope to gotowy **CoroutineScope**, który jest **wbudowany** w klasę ViewModel z biblioteki AndroidX Lifecycle. Jego głównym zadaniem jest uruchamianie korutyn, których **cykl życia** jest **automatycznie powiązany** z **cyklem życia ViewModelu**.

collectAsStateWithLifecycle to bezpieczniejsza i zoptymalizowana wersja collectAsState, która **wstrzymuje zbieranie danych ze strumienia**, gdy aplikacja **działa w tle** (jest w stanie STOPPED) i wznawia je, gdy wraca na pierwszy plan.

Uniwersytet Wrocławski

SharedFLow

Wiemy już, że StateFlow jest jak główny ekran transmisji na Twitchu – **zawsze** pokazuje **aktualny stan** transmisji, liczbę widzów itp. Nowy widz, który dołącza, od razu widzi ten stan.

Co się dzieje w sytuacji, gdy streamer dostanie donację? Na ekranie pojawia się animacja i dźwięk, a na czacie wyskakuje wiadomość. To jest **ZDARZENIE**, a **nie STAN**. Nie chcemy, żeby nowy widz, który dołączy do transmisji 5 minut później, nagle zobaczył animację tej starej donacji. To **zdarzenie jest ulotne** i przeznaczone tylko dla tych, którzy oglądali w danym momencie.

SharedFLow

Wiemy już, że StateFlow jest jak główny ekran transmisji na Twitchu – **zawsze** pokazuje **aktualny stan** transmisji, liczbę widzów itp. Nowy widz, który dołącza, od razu widzi ten stan.

Co się dzieje w sytuacji, gdy streamer dostanie donację? Na ekranie pojawia się animacja i dźwięk, a na czacie wyskakuje wiadomość. To jest **ZDARZENIE**, a **nie STAN**. Nie chcemy, żeby nowy widz, który dołączy do transmisji 5 minut później, nagle zobaczył animację tej starej donacji. To **zdarzenie jest ulotne** i przeznaczone tylko dla tych, którzy oglądali w danym momencie.

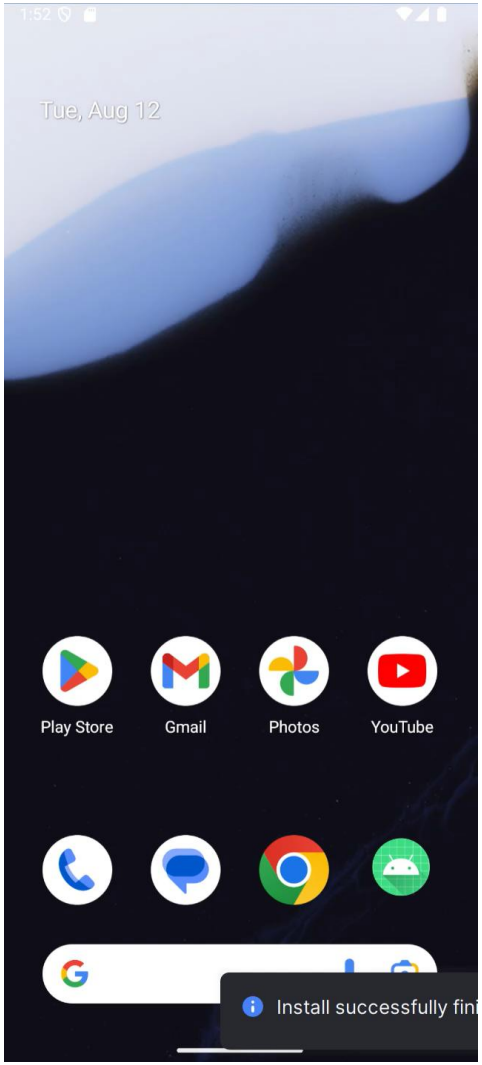
Do obsługi takich jednorazowych zdarzeń służy **SharedFlow**. To również jest ***'gorący'*** strumień, ale działający na innych zasadach niż StateFlow

SharedFLow

Do obsługi jednorazowych zdarzeń służy **SharedFlow**. To również jest ***'gorący'*** strumień, ale działający na innych zasadach niż StateFlow

Główny ekran transmisji. **Zawsze** jest widoczny, **zawsze** ma jakąś wartość (**stan**).

Powiadomienia o subskrypcji, donacji lub wiadomości na czacie. Są to **jednorazowe** "*strzały*" informacji. Pojawiają się, widzą je **wszyscy**, **którzy akurat patrzą**, i **znikają**. Nie są częścią trwałego stanu.



Uniwersytet Wrocławski

SharedFLow

Do obsługi jednorazowych zdarzeń służy **SharedFlow**. To również jest ***'gorący'*** strumień, ale działający na innych zasadach niż StateFlow

```
class TwitchStreamerEvents {
    private val _events = MutableSharedFlow<String>()
    val events: SharedFlow<String> = _events.asSharedFlow()

    // Funkcja do wysłania nowego zdarzenia do wszystkich aktualnych widzów
    suspend fun simulateDonation(donator: String) {
        val message = "💸 Donacja od '$donator'!"
        println("STREAMER: Wysyłam zdarzenie: '$message'")
        _events.emit(message)
    }
}
```

SharedFLow

Do obsługi jednorazowych zdarzeń służy **SharedFlow**. To również jest ***'gorący'*** strumień, ale działający na innych zasadach niż StateFlow

Rozpoczęcie transmisji

```
fun main() = runBlocking {
    val streamer = TwitchStreamerEvents()

    println("--- Transmisja trwa ---")

    // Widz 1 dołącza od razu i zaczyna nasłuchiwać
    val job1 = launch {
        println("😀 Widz 1 dołączył i czeka na donacje.")
        streamer.events.collect { event ->
            println("[Widz 1] Otrzymał powiadomienie: $event")
        }
    }

    delay(2000)
    streamer.simulateDonation("Anna") // Tę donację zobaczy tylko Widz 1
    delay(2000)

    // Widz 2 dołącza później
    val job2 = launch {
        println("😀 Widz 2 dołączył i czeka na donacje.")
        streamer.events.collect { event ->
            println("[Widz 2] Otrzymał powiadomienie: $event")
        }
    }

    delay(2000)
    streamer.simulateDonation("Piotr") // Tę donację zobaczą obaj widzowie
    delay(1000)

    job1.cancel()
    job2.cancel()
}
```

```
--- Transmisja trwa ---
😀 Widz 1 dołączył i czeka na donacje.
STREAMER: Wysyłam zdarzenie: '💸 Donacja od 'Anna'!'
[Widz 1] Otrzymał powiadomienie: 💸 Donacja od 'Anna'!
😀 Widz 2 dołączył i czeka na donacje.
STREAMER: Wysyłam zdarzenie: '💸 Donacja od 'Piotr'!'
[Widz 1] Otrzymał powiadomienie: 💸 Donacja od 'Piotr'!
[Widz 2] Otrzymał powiadomienie: 💸 Donacja od 'Piotr'!
```

SharedFLow

Do obsługi jednorazowych zdarzeń służy **SharedFlow**. To również jest ***'gorący'*** strumień, ale działający na innych zasadach niż StateFlow

Rozpoczęcie transmisji

Dołącza Widz 1

```
fun main() = runBlocking {
    val streamer = TwitchStreamerEvents()

    println("--- Transmisja trwa ---")

    // Widz 1 dołącza od razu i zaczyna nasłuchiwać
    val job1 = launch {
        println("😀 Widz 1 dołączył i czeka na donacje.")
        streamer.events.collect { event ->
            println("[Widz 1] Otrzymał powiadomienie: $event")
        }
    }

    delay(2000)
    streamer.simulateDonation("Anna") // Tę donację zobaczy tylko Widz 1
    delay(2000)

    // Widz 2 dołącza później
    val job2 = launch {
        println("😀 Widz 2 dołączył i czeka na donacje.")
        streamer.events.collect { event ->
            println("[Widz 2] Otrzymał powiadomienie: $event")
        }
    }

    delay(2000)
    streamer.simulateDonation("Piotr") // Tę donację zobaczą obaj widzowie
    delay(1000)

    job1.cancel()
    job2.cancel()
}
```

```
--- Transmisja trwa ---
😀 Widz 1 dołączył i czeka na donacje.
STREAMER: Wysyłam zdarzenie: '💸 Donacja od 'Anna'!'
[Widz 1] Otrzymał powiadomienie: 💸 Donacja od 'Anna'!
😀 Widz 2 dołączył i czeka na donacje.
STREAMER: Wysyłam zdarzenie: '💸 Donacja od 'Piotr'!'
[Widz 1] Otrzymał powiadomienie: 💸 Donacja od 'Piotr'!
[Widz 2] Otrzymał powiadomienie: 💸 Donacja od 'Piotr'!
```

SharedFLow

Do obsługi jednorazowych zdarzeń służy **SharedFlow**. To również jest ***'gorący'*** strumień, ale działający na innych zasadach niż StateFlow

Rozpoczęcie transmisji

Dołącza Widz 1

Donacja, którą widzi tylko Widz 1

```
fun main() = runBlocking {
    val streamer = TwitchStreamerEvents()

    println("--- Transmisja trwa ---")

    // Widz 1 dołącza od razu i zaczyna nasłuchiwać
    val job1 = launch {
        println("😀 Widz 1 dołączył i czeka na donacje.")
        streamer.events.collect { event ->
            println("[Widz 1] Otrzymał powiadomienie: $event")
        }
    }

    delay(2000)
    streamer.simulateDonation("Anna") // Tę donację zobaczy tylko Widz 1
    delay(2000)

    // Widz 2 dołącza później
    val job2 = launch {
        println("😀 Widz 2 dołączył i czeka na donacje.")
        streamer.events.collect { event ->
            println("[Widz 2] Otrzymał powiadomienie: $event")
        }
    }

    delay(2000)
    streamer.simulateDonation("Piotr") // Tę donację zobaczą obaj widzowie
    delay(1000)

    job1.cancel()
    job2.cancel()
}
```

```
--- Transmisja trwa ---
😀 Widz 1 dołączył i czeka na donacje.
STREAMER: Wysyłam zdarzenie: '💸 Donacja od 'Anna'!'
[Widz 1] Otrzymał powiadomienie: 💸 Donacja od 'Anna'!
😀 Widz 2 dołączył i czeka na donacje.
STREAMER: Wysyłam zdarzenie: '💸 Donacja od 'Piotr'!'
[Widz 1] Otrzymał powiadomienie: 💸 Donacja od 'Piotr'!
[Widz 2] Otrzymał powiadomienie: 💸 Donacja od 'Piotr'!
```

SharedFLow

Do obsługi jednorazowych zdarzeń służy **SharedFlow**. To również jest ***'gorący'*** strumień, ale działający na innych zasadach niż StateFlow

Rozpoczęcie transmisji

Dołącza Widz 1

Donacja, którą widzi tylko Widz 1

Dołącza Widz 2

```
fun main() = runBlocking {
    val streamer = TwitchStreamerEvents()

    println("--- Transmisja trwa ---")

    // Widz 1 dołącza od razu i zaczyna nasłuchiwać
    val job1 = launch {
        println("😀 Widz 1 dołączył i czeka na donacje.")
        streamer.events.collect { event ->
            println("[Widz 1] Otrzymał powiadomienie: $event")
        }
    }

    delay(2000)
    streamer.simulateDonation("Anna") // Tę donację zobaczy tylko Widz 1
    delay(2000)

    // Widz 2 dołącza później
    val job2 = launch {
        println("😀 Widz 2 dołączył i czeka na donacje.")
        streamer.events.collect { event ->
            println("[Widz 2] Otrzymał powiadomienie: $event")
        }
    }

    delay(2000)
    streamer.simulateDonation("Piotr") // Tę donację zobaczą obaj widzowie
    delay(1000)

    job1.cancel()
    job2.cancel()
}
```

```
--- Transmisja trwa ---
😀 Widz 1 dołączył i czeka na donacje.
STREAMER: Wysyłam zdarzenie: '💸 Donacja od 'Anna'!'
[Widz 1] Otrzymał powiadomienie: 💸 Donacja od 'Anna'!
😀 Widz 2 dołączył i czeka na donacje.
STREAMER: Wysyłam zdarzenie: '💸 Donacja od 'Piotr'!'
[Widz 1] Otrzymał powiadomienie: 💸 Donacja od 'Piotr'!
[Widz 2] Otrzymał powiadomienie: 💸 Donacja od 'Piotr'!
```

SharedFLow

Do obsługi jednorazowych zdarzeń służy **SharedFlow**. To również jest ***'gorący'*** strumień, ale działający na innych zasadach niż StateFlow

Rozpoczęcie transmisji

Dołącza Widz 1

Donacja, którą widzi tylko Widz 1

Dołącza Widz 2

Donacja, którą widzi Widz 1 i Widz 2

```
fun main() = runBlocking {
    val streamer = TwitchStreamerEvents()

    println("--- Transmisja trwa ---")

    // Widz 1 dołącza od razu i zaczyna nasłuchiwać
    val job1 = launch {
        println("😀 Widz 1 dołączył i czeka na donacje.")
        streamer.events.collect { event ->
            println("[Widz 1] Otrzymał powiadomienie: $event")
        }
    }

    delay(2000)
    streamer.simulateDonation("Anna") // Tę donację zobaczy tylko Widz 1
    delay(2000)

    // Widz 2 dołącza później
    val job2 = launch {
        println("😀 Widz 2 dołączył i czeka na donacje.")
        streamer.events.collect { event ->
            println("[Widz 2] Otrzymał powiadomienie: $event")
        }
    }

    delay(2000)
    streamer.simulateDonation("Piotr") // Tę donację zobaczą obaj widzowie
    delay(1000)

    job1.cancel()
    job2.cancel()
}
```

```
--- Transmisja trwa ---
😀 Widz 1 dołączył i czeka na donacje.
STREAMER: Wysyłam zdarzenie: '💸 Donacja od 'Anna'!'
[Widz 1] Otrzymał powiadomienie: 💸 Donacja od 'Anna'!
😀 Widz 2 dołączył i czeka na donacje.
STREAMER: Wysyłam zdarzenie: '💸 Donacja od 'Piotr'!'
[Widz 1] Otrzymał powiadomienie: 💸 Donacja od 'Piotr'!
[Widz 2] Otrzymał powiadomienie: 💸 Donacja od 'Piotr'!
```

SharedFLow

Do obsługi jednorazowych zdarzeń służy **SharedFlow**. To również jest ***'gorący'*** strumień, ale działający na innych zasadach niż StateFlow

Rozpoczęcie transmisji

Dołącza Widz 1

Donacja, którą widzi tylko Widz 1

Dołącza Widz 2

Donacja, którą widzi Widz 1 i Widz 2

```
fun main() = runBlocking {
    val streamer = TwitchStreamerEvents()

    println("--- Transmisja trwa ---")

    // Widz 1 dołącza od razu i zaczyna nasłuchiwać
    val job1 = launch {
        println("😀 Widz 1 dołączył i czeka na donacje.")
        streamer.events.collect { event ->
            println("[Widz 1] Otrzymał powiadomienie: $event")
        }
    }

    delay(2000)
    streamer.simulateDonation("Anna") // Tę donację zobaczy tylko Widz 1
    delay(2000)

    // Widz 2 dołącza później
    val job2 = launch {
        println("😀 Widz 2 dołączył i czeka na donacje.")
        streamer.events.collect { event ->
            println("[Widz 2] Otrzymał powiadomienie: $event")
        }
    }

    delay(2000)
    streamer.simulateDonation("Piotr") // Tę donację zobaczą obaj widzowie
    delay(1000)

    job1.cancel()
    job2.cancel()
}
```

```
--- Transmisja trwa ---
😀 Widz 1 dołączył i czeka na donacje.
STREAMER: Wysyłam zdarzenie: '💸 Donacja od 'Anna'!'
[Widz 1] Otrzymał powiadomienie: 💸 Donacja od 'Anna'!
😀 Widz 2 dołączył i czeka na donacje.
STREAMER: Wysyłam zdarzenie: '💸 Donacja od 'Piotr'!'
[Widz 1] Otrzymał powiadomienie: 💸 Donacja od 'Piotr'!
[Widz 2] Otrzymał powiadomienie: 💸 Donacja od 'Piotr'!
```

Widz 2 nigdy **nie otrzymał powiadomienia** od Anny, ponieważ dołączył do transmisji **po tym, jak to zdarzenie już miało miejsce**.

Uniwersytet Wrocławski

SharedFLow

```
class StreamerViewModel : ViewModel() {
    // STAN: Liczba widzów (StateFlow)
    2 Usages
    private val _viewerCount = MutableStateFlow( value = 0)
    1 Usage
    val viewerCount: StateFlow<Int> = _viewerCount.asStateFlow()

    // ZDARZENIA: Powiadomienia o donacjach (SharedFlow)
    2 Usages
    private val _events = MutableSharedFlow<String>()
    2 Usages
    val events: SharedFlow<String> = _events.asSharedFlow()

    init {
        viewModelScope.launch {
            while (true) {
                delay( timeMillis = 2000)
                _viewerCount.value = Random.nextInt( from = 100,  until = 1000)
            }
        }
    }

    1 Usage
    fun simulateDonation() {
        viewModelScope.launch {
            _events.emit( value = "Nowa donacja od 'Widz123'!")
        }
    }
}
```

Uniwersytet Wrocławski

SharedFLow

```
@OptIn( ...markerClass = ExperimentalMaterial3Api::class)
@Composable
fun StreamerScreen(viewModel: StreamerViewModel = viewModel()) {
    // Subskrypcja STANU (StateFlow) - bez zmian
    val viewerCount by viewModel.viewerCount.collectAsStateWithLifecycle()

    // Stany dla dwóch niezależnych widzów
    var viewer1Event by remember { mutableStateOf( value = "Widz 1: Ogląda stream...") }
    var viewer2Event by remember { mutableStateOf( value = "Widz 2: Ogląda stream...") }

    // Uruchamiamy DWA niezależne odbiory tego samego strumienia zdarzeń
    LaunchedEffect( key1 = Unit) {
        // Korutyna dla Widza 1
        launch {
            viewModel.events.collect { message ->
                viewer1Event = "Widz 1: Otrzymał '$message'"
            }
        }
        // Korutyna dla Widza 2
        launch {
            viewModel.events.collect { message ->
                viewer2Event = "Widz 2: Otrzymał '$message'"
            }
        }
    }
```

Uniwersytet Wrocławski

Podsumowanie

Narzędzie	Analogia	Przeznaczenie	Kluczowa Cecha
Flow	**Film na YouTube** 📺	Jednorazowe operacje, które zwracają sekwencję danych.	**Zimny** - każdy widz ogląda od nowa.
StateFlow	**Główny ekran transmisji na Twitchu** 🎮	Reprezentowanie **stanu** UI, który musi być zawsze dostępny.	**Gorący** - zawsze ma wartość, nowi widzowie widzą aktualny stan.
SharedFlow	**Alerty / Czat na Twitchu** 💬	Wysyłanie **jednorazowych zdarzeń**, które nie powinny być odtwarzane.	**Gorący** - bezstanowy, wysyła "strzał" informacji do obecnych widzów.

Uniwersytet Wrocławski

Przykład zastosowania