Uniwersytet Wrocławski

Rafał Lewandków

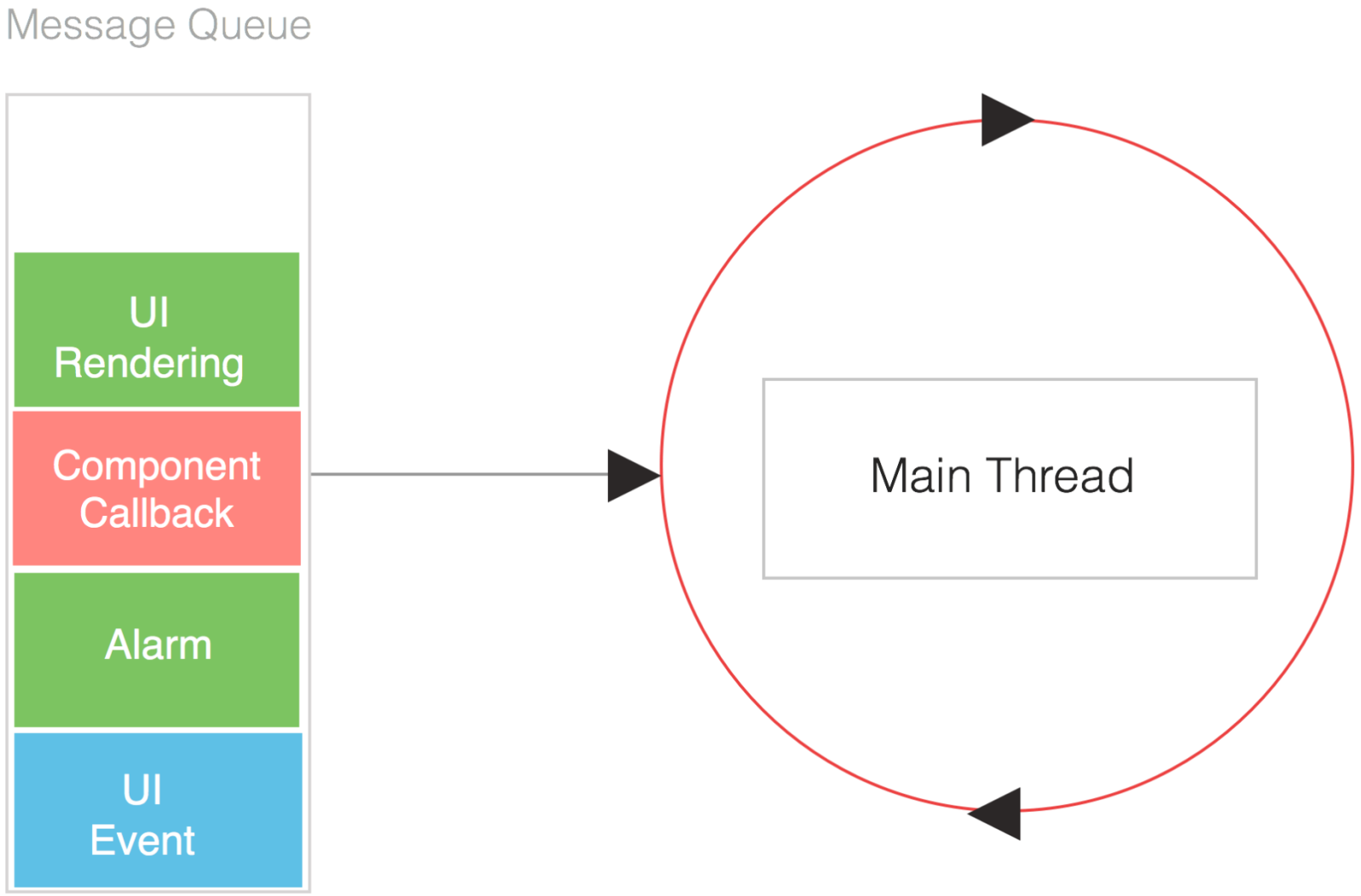
PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

WYKŁAD 9

Strumienie danych

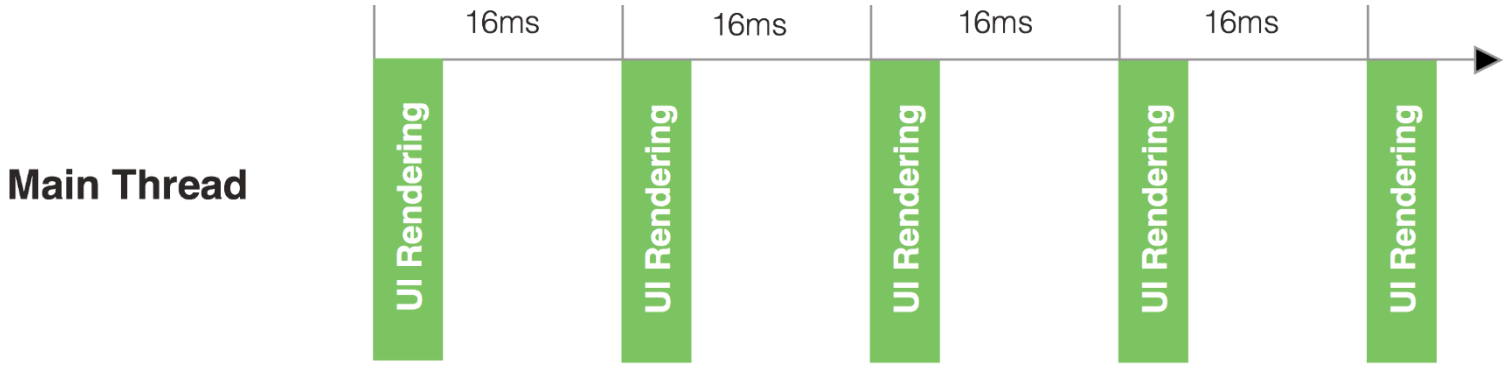
- Asynchroniczne przetwarzanie danych
- LiveData
- Flow
- StateFlow
- SharedFlow
- ComposeState

Wątek główny



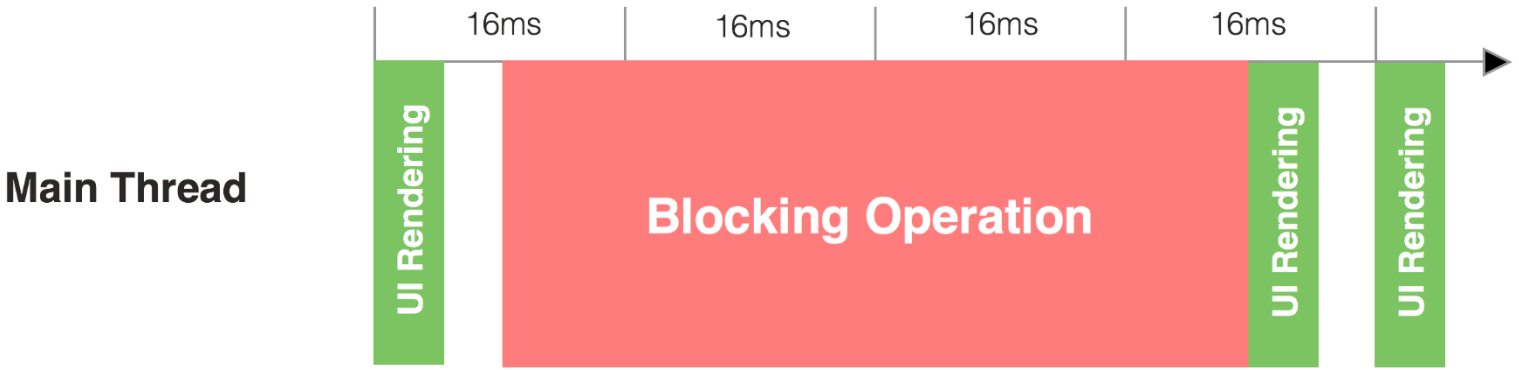
https://hvasconcelos.github.io/articles/Offloading-work-from-the-UI-Thread

Wątek główny



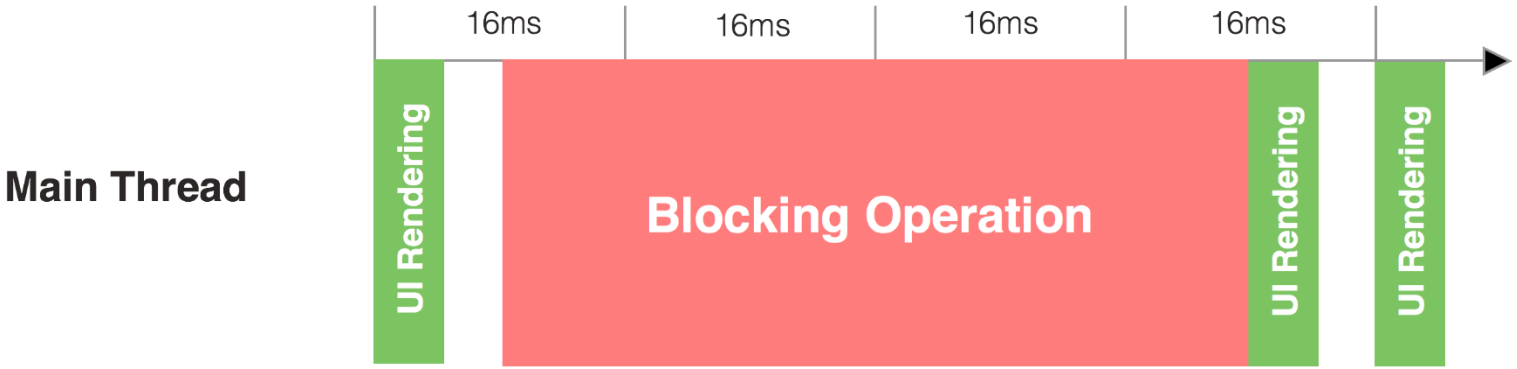
https://hvasconcelos.github.io/articles/Offloading-work-from-the-UI-Thread

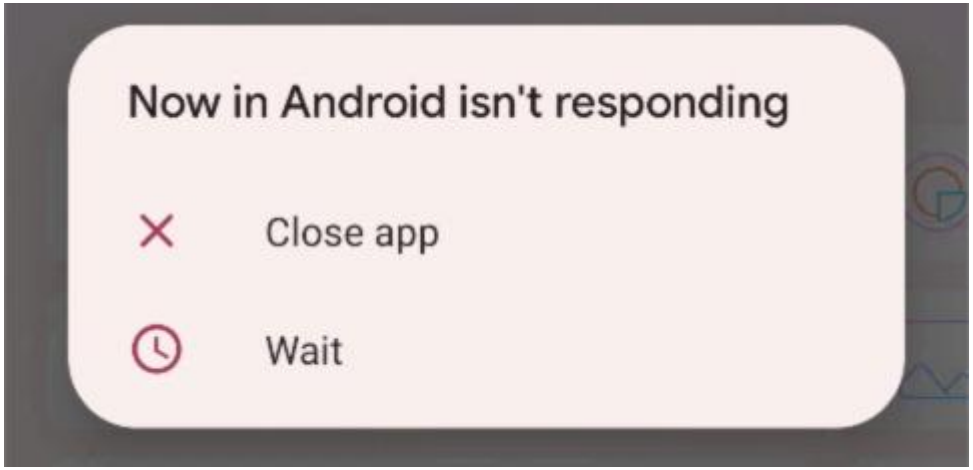
Wątek główny



https://hvasconcelos.github.io/articles/Offloading-work-from-the-UI-Thread

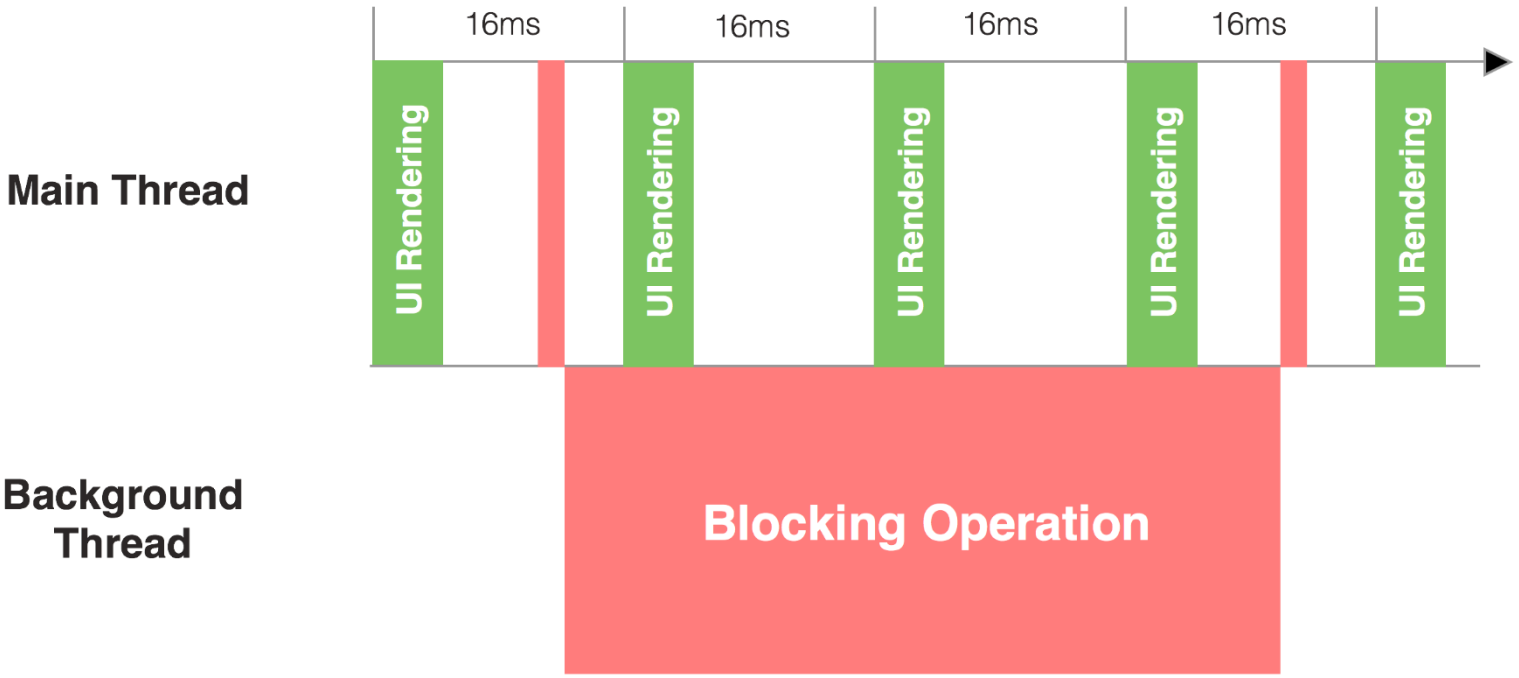
Wątek główny





https://hvasconcelos.github.io/articles/Offloading-work-from-the-UI-Thread

Wątek główny



https://hvasconcelos.github.io/articles/Offloading-work-from-the-UI-Thread

Wątek główny

- **Network calls**
- **Database queries**
- **Długie obliczenia**
- **Taski działające w tle**

https://hvasconcelos.github.io/articles/Offloading-work-from-the-UI-Thread

Uniwersytet Wrocławski

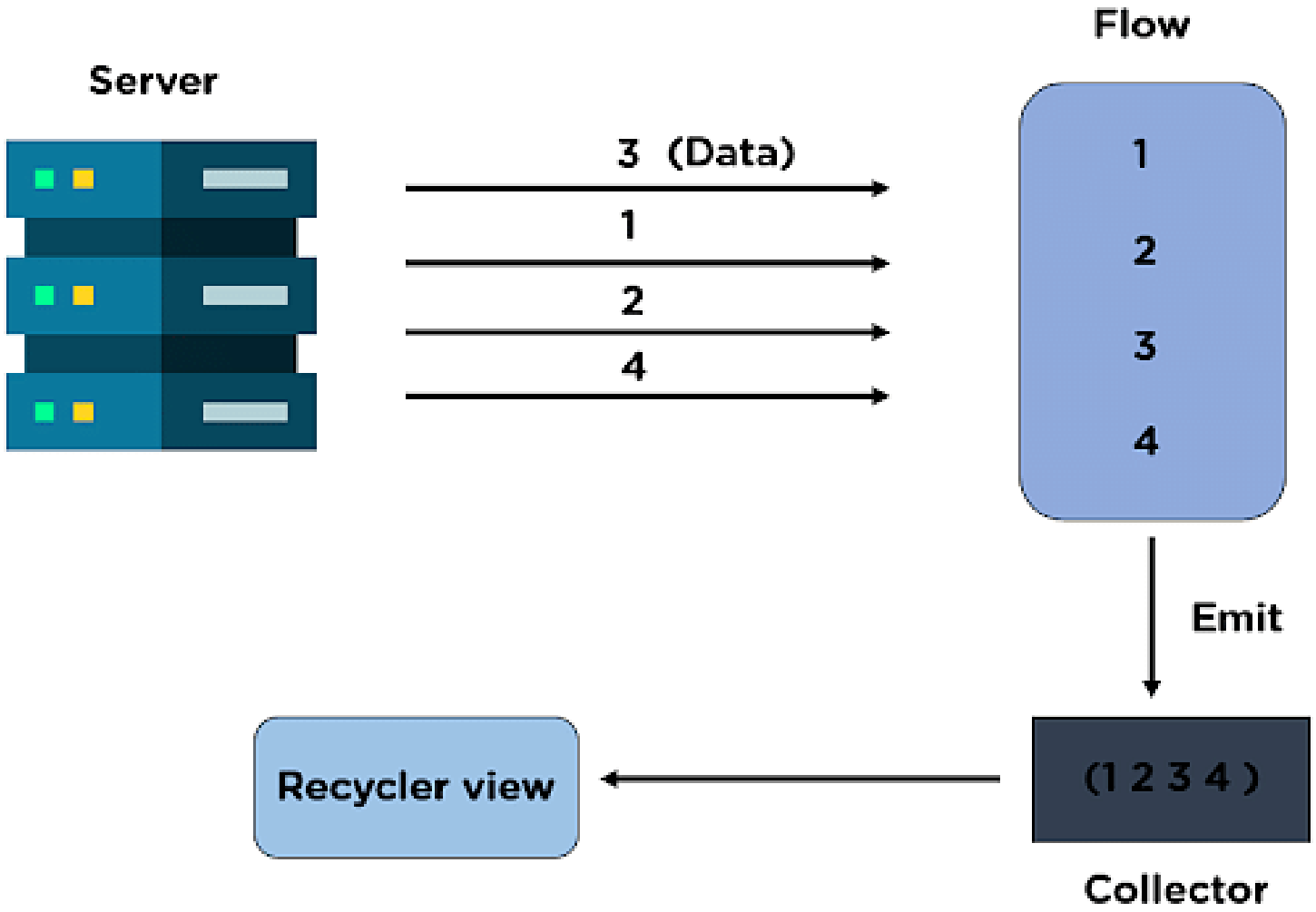
Data stream

Strumienie danych (ang. *data streams*) to sekwencje wartości lub zdarzeń, które są emitowane w określonym czasie lub w odpowiedzi na różne zdarzenia. Strumienie danych mogą być dynamicznie aktualizowane i przetwarzane.

Główne cechy strumieni danych to:

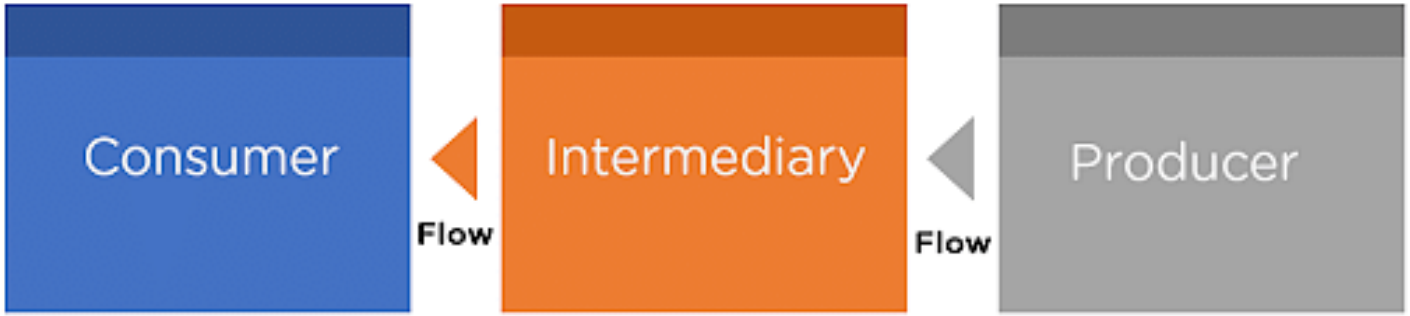
- Emitowanie wartości: Strumienie danych mogą emitować wartości w czasie, jedną po drugiej. Emitowanie wartości może być synchroniczne lub asynchroniczne.
- Reaktywne odbieranie: Odbiorcy strumienia mogą reagować na emitowane wartości, podejmując odpowiednie działania. Odbieranie może odbywać się w czasie rzeczywistym lub w reakcji na określone zdarzenia.
- Przetwarzanie strumienia: Strumienie danych mogą być przetwarzane i transformowane za pomocą różnych operacji, takich jak filtrowanie, mapowanie, łączenie, grupowanie itp.
- Obsługa błędów i anulowanie: Strumienie danych mogą obsługiwać sytuacje błędne i obsługiwać anulowanie strumienia w dowolnym momencie.

Flow



https://www.simplilearn.com/tutorials/kotlin-tutorial/an-ultimate-guide-to-kotlin-flows

Flow



https://www.simplilearn.com/tutorials/kotlin-tutorial/an-ultimate-guide-to-kotlin-flows

Flow


```
object DataProvider {
    val data: List<String> = listOf(
        "pierwszy",
        "drugi",
        "trzeci",
        "czwarty",
        "piąty",
        "szósty",
        "siódmy",
        "ósmy",
        "dziewiąty",
        "dziesiąty",
        "jedenasty",
        "dwunasty",
        "trzynasty",
        "czternasty",
        "piętnasty",
        "szesnasty",
        "siedemnasty",
        "osiemnasty",
        "dziewiętnasty",
        "dwudziesty",
    )
}
```

Flow


```
class WordsViewModel : ViewModel() {
    val wordsFlow = flow{
        var index = 0
        while (index < DataProvider.data.size){
            emit(DataProvider.data[index])
            delay(500L)
            index++
        }
    }
}
```

Uniwersytet Wrocławski

Flow + Fragment/Aktywność


```
class WordsViewModel : ViewModel() {
    val wordsFlow = flow{
        var index = 0
        while (index < DataProvider.data.size){
            emit(DataProvider.data[index])
            delay(500L)
            index++
        }
    }
}
```

```
viewLifecycleOwner.lifecycleScope.launch {
    viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED){
        viewModel.wordsFlow.collect{ word ->
            binding.wordText.text = word
        }
    }
}
```

Uniwersytet Wrocławski

Flow + Compose


```
class WordsViewModel : ViewModel() {
    val wordsFlow = flow{
        var index = 0
        while (index < DataProvider.data.size){
            emit(DataProvider.data[index])
            delay(500L)
            index++
        }
```

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            FlowBasicsComposeTheme {

                val viewModel: WordsViewModel = viewModel()

                val word = viewModel.wordsFlow.collectAsStateWithLifecycle("start")

                Surface(
                    modifier = Modifier.fillMaxSize(),
                    color = MaterialTheme.colorScheme.background
                ) {
                    Column(
                        modifier = Modifier.fillMaxSize(),
                        verticalArrangement = Arrangement.Center,
                        horizontalAlignment = Alignment.CenterHorizontally
                    ){
                        Text(
                            text = word.value, // ustawienie wartości w polu Text
                            fontSize = 56.sp,
                            modifier = Modifier.fillMaxWidth(),
                            textAlign = TextAlign.Center
                        )
```

Uniwersytet Wrocławski

Flow vs LiveData

Kotlin Flow	LiveData
• As flow is specific to kotlin language, hence it supports multi-platform.	• Where as LiveData is built only for Android platform, hence it does not supports multi-platform.
• StateFlow requires an initial value	• LiveData does not require any intial value.
• Flow collection is not stopped automatically, but this behaviour can be easily achieved with the repeatOnLifeCycle extension.	• The method observe() from LiveData automatically unregisters the consumer when the view enters the STOPPED state.
• With Flow as return type, room created a new possibility of seamless data integration across the app between database and UI without writing any extra code.	• Lack of seamless data integration across between database and UI especially using Room.
• By using Shared Flow we can avoid fecthing latest values when configuration changes.	• LiveData always fetch latest values when configuration changes it might be bad when we use snack bar for users.

https://www.google.com/url?sa=i&url=https%3A%2F%2Fblogs.halodoc.io%2Freactive-way-of-mobile-app-development-using-kotlin-flow%2F

Uniwersytet Wrocławski

Hot Flow vs Cold Flow

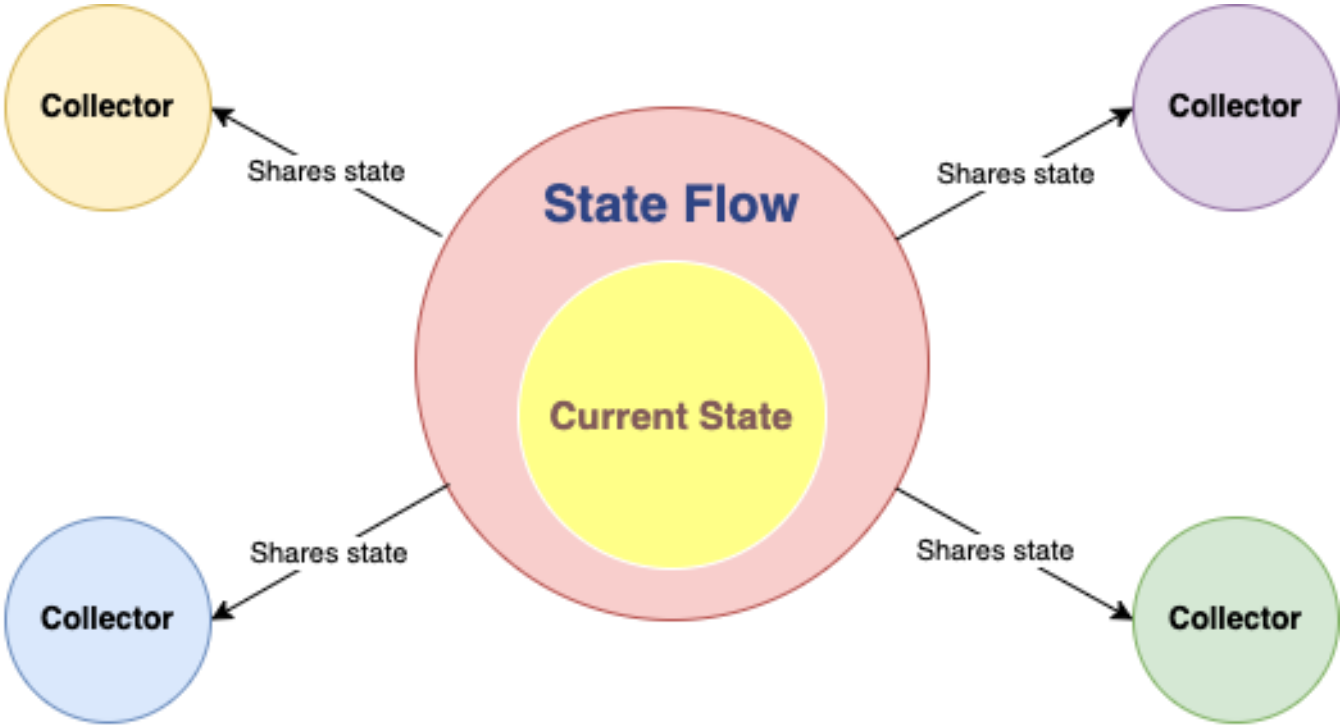
Cold Flow	Hot Flow
It emits data only when there is a collector.	It emits data even when there is no collector.
It does not store data.	It can store data.
It can't have multiple collectors.	It can have multiple collectors.

- **Cold Flow**: Zimny strumień emituje wartości tylko wtedy, gdy istnieje aktywny collector. Każdy collector odbiera emitowane wartości niezależnie, i zaczynają otrzymywać emisje od początku, gdy się zapiszą.
- **Hot Flow**: Gorący strumień emituje wartości niezależnie od tego, czy są aktywni collectors. Może generować wartości nawet wtedy, gdy nie ma subskrybentów. Nowi zbieracze dołączający do gorącego strumienia mogą przegapić emisje, które wystąpiły przed rozpoczęciem ich nasłuchiwania.

StateFlow	SharedFlow
Cold Flow	Hot Flow

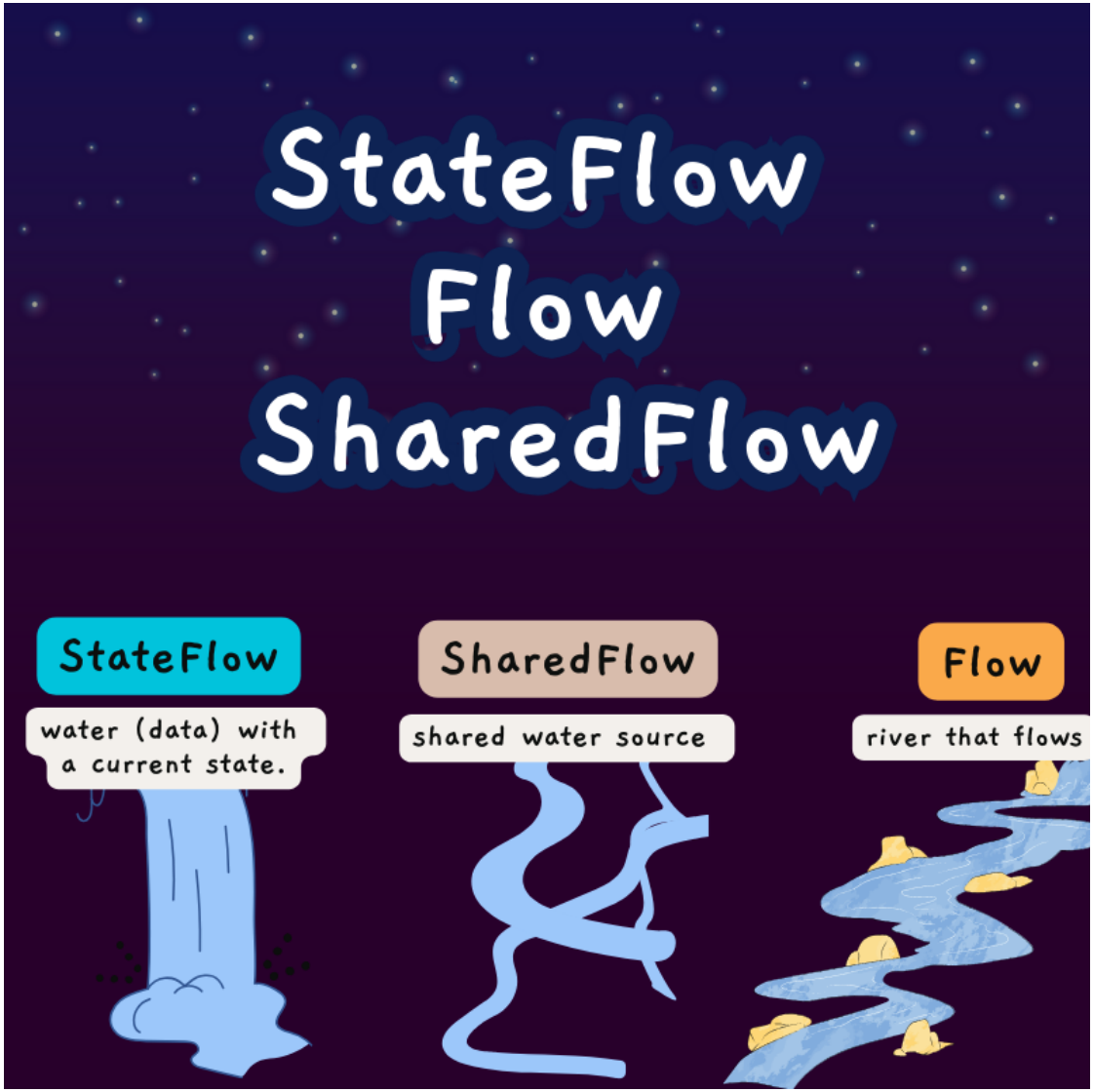
StateFlow

Flow to ogólna koncepcja reprezentująca sekwencję wartości w czasie. StateFlow to specyficzny typ Flow, który reprezentuje wartość z bieżącym stanem.



https://www.google.com/url?sa=i&url=https%3A%2F%2Fmedium.com%2F%40mortitech%2Fsharedflow-vs-stateflow-a-comprehensive-guide-to-kotlin-flows-503576b4de31

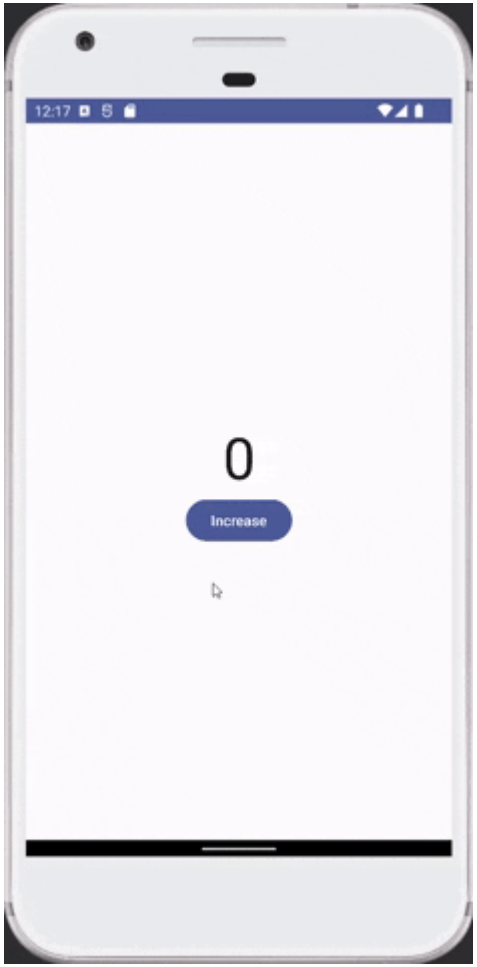
StateFlow



https://www.google.com/url?sa=i&url=https%3A%2F%2Fpinarturgut09.medium.com%2Fdifference-between-flow-shared-flow-and-stateflow-in-android-e03b2a3e5ffc

Uniwersytet Wrocławski

StateFlow



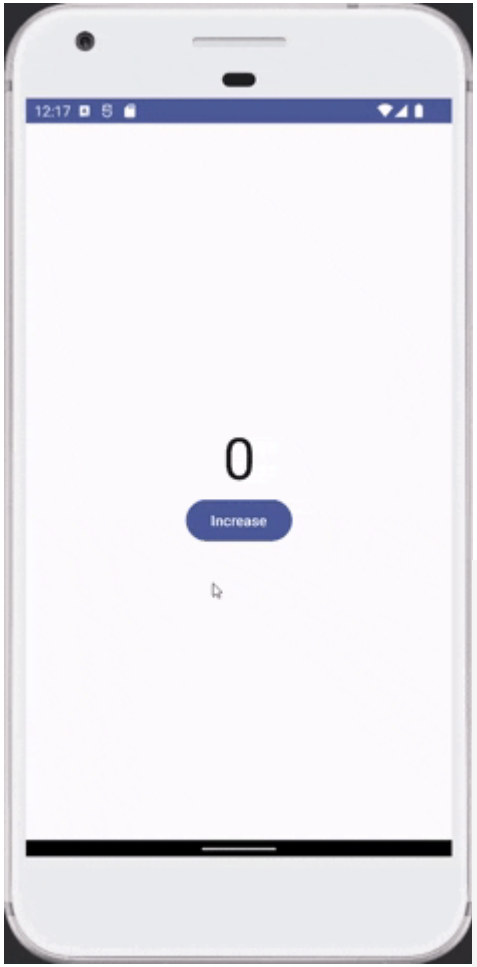

```
class CounterViewModel : ViewModel() {

    private val _stateFlow = MutableStateFlow(0)
    val stateFlow = _stateFlow.asStateFlow()

    fun increase(){
        _stateFlow.value += 1
    }
}
```

Uniwersytet Wrocławski

StateFlow + Fragment/Aktywność




```
class CounterViewModel : ViewModel() {

    private val _stateFlow = MutableStateFlow(0)
    val stateFlow = _stateFlow.asStateFlow()

    fun increase(){
        _stateFlow.value += 1
    }
}
```

```
viewLifecycleOwner.lifecycleScope.launch {
    viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED){
        viewModel.stateFlow.collectLatest{ counter ->
            binding.counterText.text = counter.toString()
        }
    }
}

binding.increaseButton.setOnClickListener {
    viewModel.increase()
}
```

Uniwersytet Wrocławski

StateFlow + Compose


```
class CounterViewModel : ViewModel() {

    private val _stateFlow = MutableStateFlow(0)
    val stateFlow = _stateFlow.asStateFlow()

    fun increase(){
        _stateFlow.value += 1
    }
}
```

```
val viewModel: CounterViewModel = viewModel()  // tworzymy instancję viewmodel
val counter = viewModel.stateFlow.collectAsStateWithLifecycle(0) // tworzymy pole typu State

Surface(
    modifier = Modifier.fillMaxSize(),
    color = MaterialTheme.colorScheme.background
) {
    Column(
        modifier = Modifier.fillMaxSize(),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment = Alignment.CenterHorizontally
    ){
        Text(
            text = counter.value.toString(), // ustawiamy wartość
            fontSize = 56.sp,
            modifier = Modifier.fillMaxWidth(),
            textAlign = TextAlign.Center
        )
        Button(onClick = { viewModel.increase() }) { // Wywołujemy funkcję increase() po naci
            Text(text = "Increase")
        }
    }
}
```

Uniwersytet Wrocławski

SharedFlow

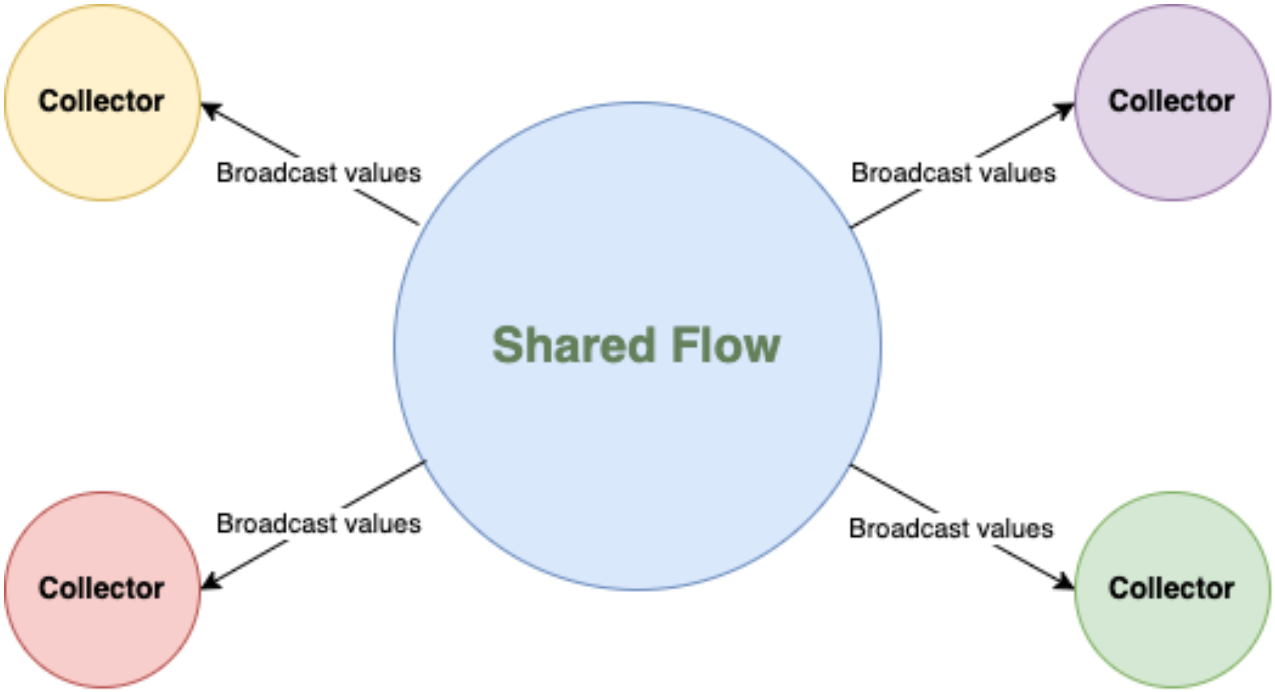
Główne cechy `SharedFlow`:

- Współdzielone wartości: Umożliwia emitowanie wartości do wielu obserwatorów. Wszyscy subskrybenci otrzymują te same emitowane wartości, dzięki czemu można synchronizować dane między różnymi komponentami w aplikacji.
- Zachowanie bufora: Może przechowywać określoną ilość ostatnich emitowanych wartości, co pozwala subskrybentom otrzymać te wartości po dołączeniu do strumienia. Można skonfigurować rozmiar bufora przy tworzeniu SharedFlow.

SharedFlow

Główne cechy `SharedFlow`:

- Współdzielone wartości: Umożliwia emitowanie wartości do wielu obserwatorów. Wszyscy subskrybenci otrzymują te same emitowane wartości, dzięki czemu można synchronizować dane między różnymi komponentami w aplikacji.
- Zachowanie bufora: Może przechowywać określoną ilość ostatnich emitowanych wartości, co pozwala subskrybentom otrzymać te wartości po dołączeniu do strumienia. Można skonfigurować rozmiar bufora przy tworzeniu SharedFlow.



SharedFlow

```
class NumberViewModel : ViewModel() {

    private var number = 0
    private val _sharedFlow = MutableSharedFlow<Int>(1)
    val sharedFlow: SharedFlow<Int> = _sharedFlow

    init {
        viewModelScope.launch {
            while (true) {
                _sharedFlow.emit(number++)
                delay(500L)
            }
        }
    }
}
```

```
viewLifecycleOwner.lifecycleScope.launch {
    viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED){
        viewModel.sharedFlow.collect{ number ->
            binding.numberText.text = number.toString()
        }
    }
}
```

Uniwersytet Wrocławski

SharedFlow

```
class NumberViewModel : ViewModel() {

    private var number = 0
    private val _sharedFlow = MutableSharedFlow<Int>(1)
    val sharedFlow: SharedFlow<Int> = _sharedFlow

    init {
        viewModelScope.launch {
            while (true) {
                _sharedFlow.emit(number++)
                delay(500L)
            }
        }
    }
}
```

```
SharedFlowBasicsTheme {
    val viewModel: NumberViewModel = viewModel()
    var number by remember {
        mutableStateOf(0)
    }

    LaunchedEffect(key1 = true){
        viewModel.sharedFlow.collect{ collectedNumber ->
            number = collectedNumber
        }
    }

    Surface(
        modifier = Modifier.fillMaxSize(),
        color = MaterialTheme.colorScheme.background
    ) {
        Column(
            modifier = Modifier.fillMaxSize(),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ){
            Text(
                text = number.toString(),
                fontSize = 56.sp,
                modifier = Modifier.fillMaxWidth(),
                textAlign = TextAlign.Center
            )
```

Uniwersytet Wrocławski

StateFlow + ROOM

```
class UserViewModel(application: Application) : AndroidViewModel(application) {

    private val repository: UserRepository
    private val _usersState = MutableStateFlow<List<User>>(emptyList())
    val usersState: StateFlow<List<User>>
        get() = _usersState

    init {
        val db = UserDatabase.getDatabase(application)
        val dao = db.userDao()
        repository = UserRepository(dao)

        fetchUsers()
    }

    private fun fetchUsers() {
        viewModelScope.launch {
            repository.getUsers().collect { users ->
                _usersState.value = users
            }
        }
    }

    fun clearUsers() {
        viewModelScope.launch {
            repository.clear()
        }
    }

    fun addUser(user: User) {
        viewModelScope.launch {
            repository.add(user)
        }
    }
}
```

Uniwersytet Wrocławski

Flow -> StateFlow

```
class CounterViewModel : ViewModel() {
    private var currentVal = 0

    val counter = flow {
        while (true){
            delay(500L)
            emit(currentVal++)
        }
    }.stateIn(
        viewModelScope,
        SharingStarted.WhileSubscribed(),
        0
    )
}
```