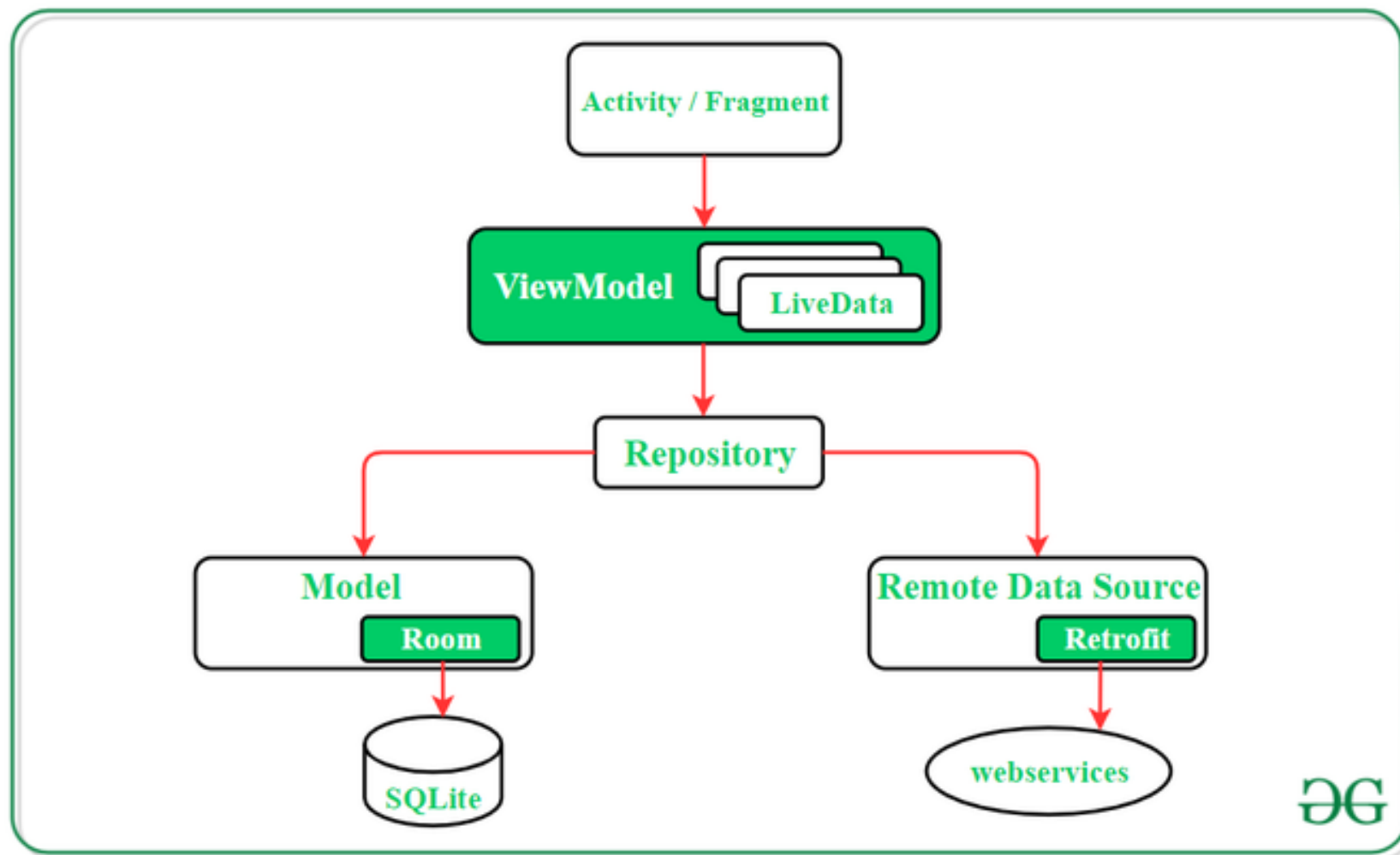




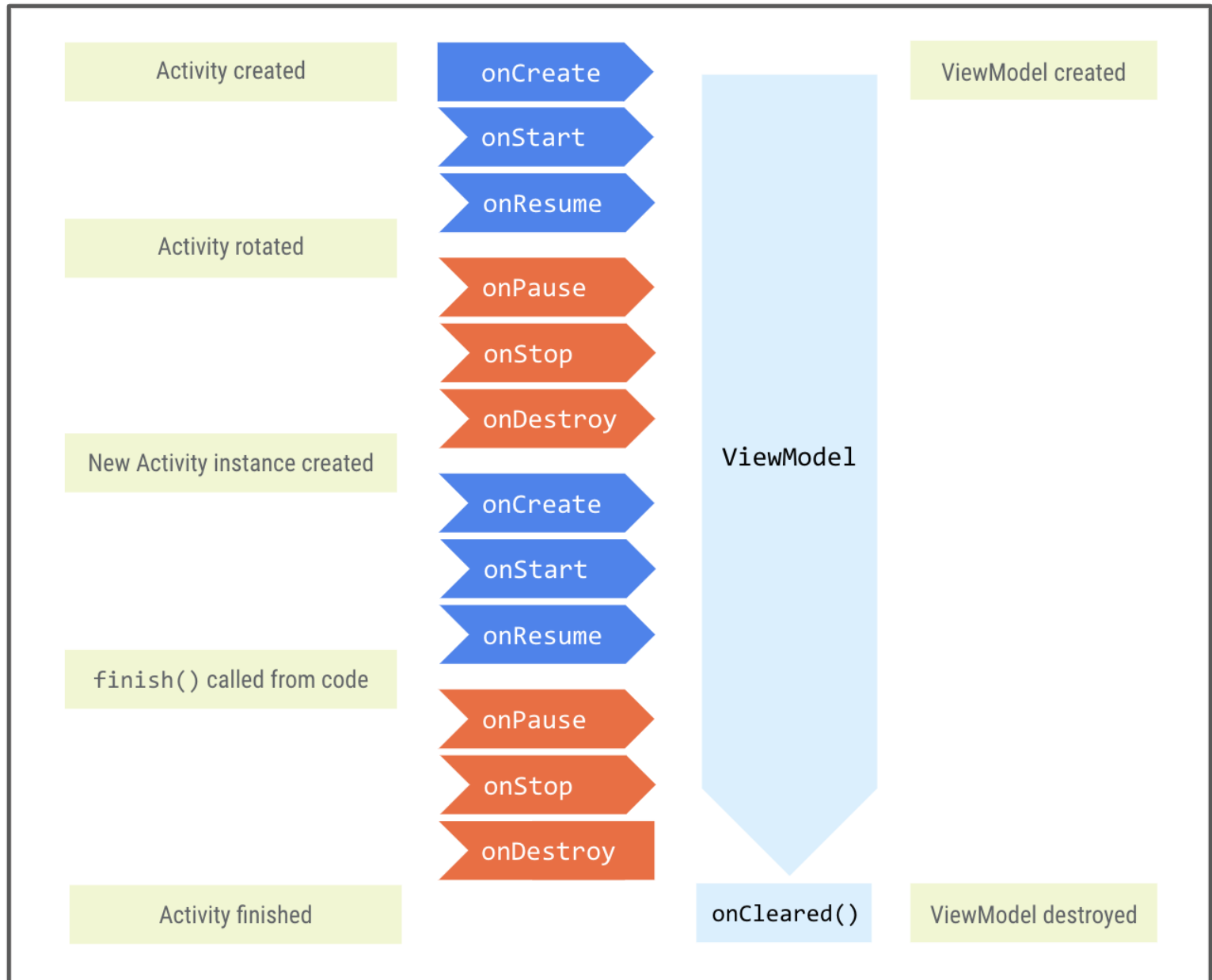
PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

WYKŁAD 7 Architektura Aplikacji 1

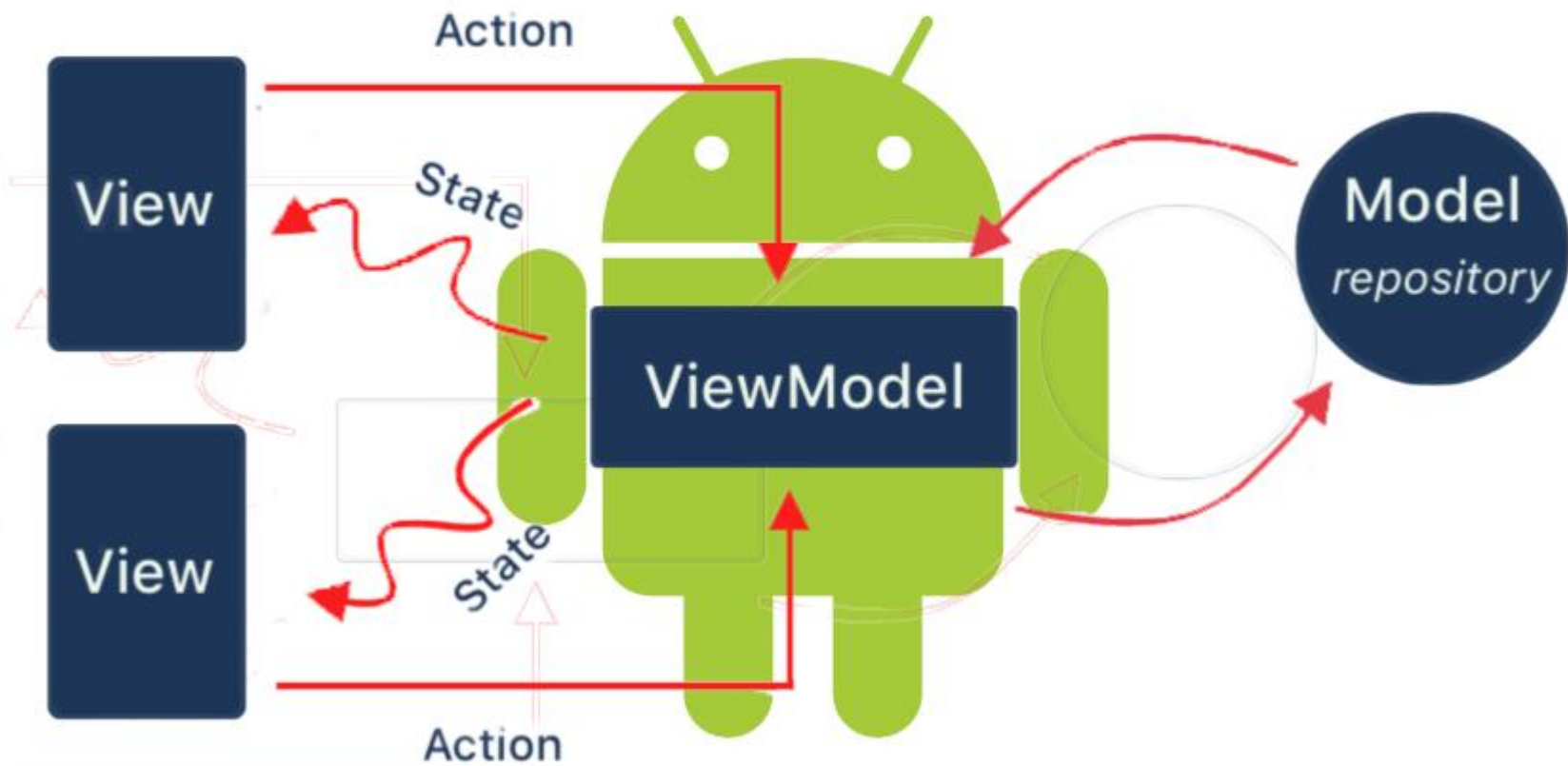
- ViewModel
- LiveData



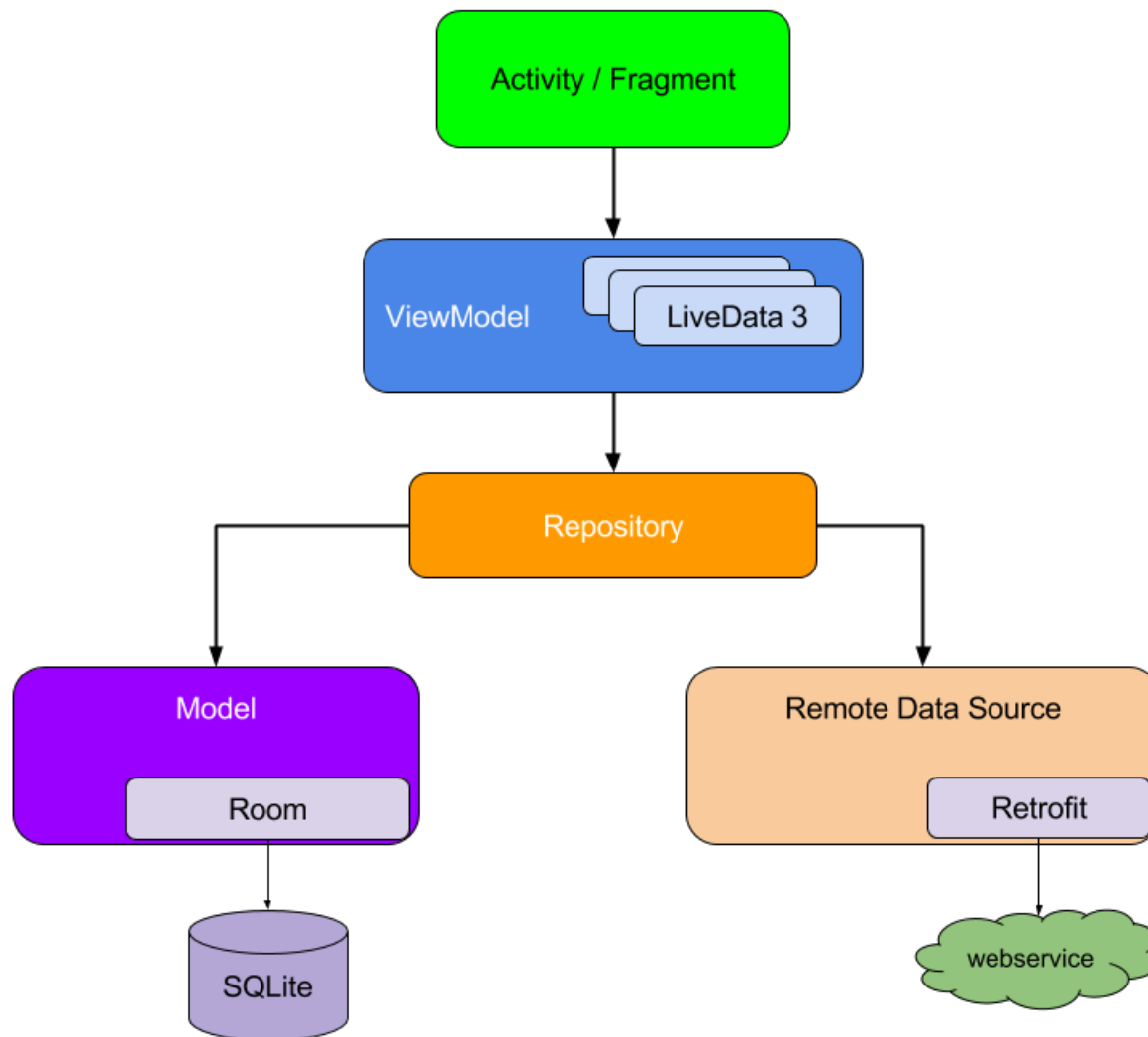
ViewModel



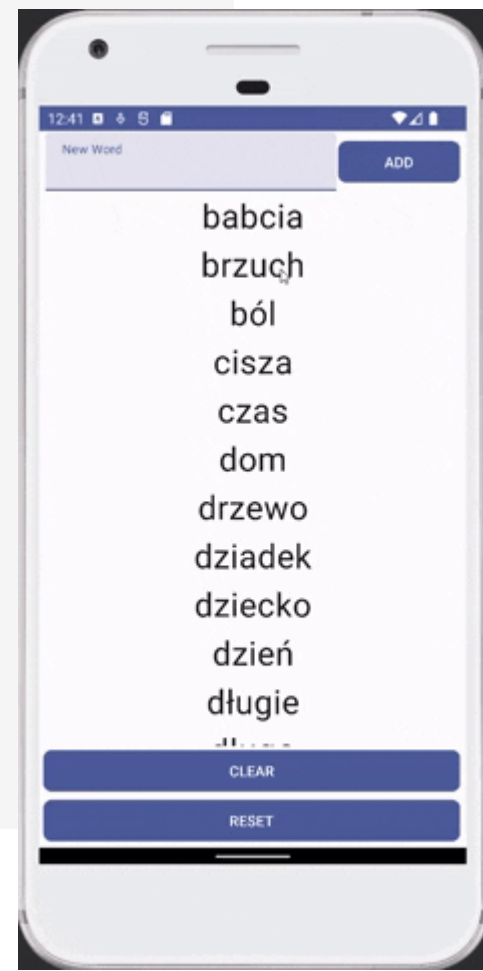
ViewModel to wzorec projektowy stosowany w programowaniu, szczególnie w kontekście tworzenia aplikacji z interfejsem użytkownika. Celem **ViewModel** jest oddzielenie logiki biznesowej aplikacji od jej warstwy prezentacji.



ViewModel to wzorec projektowy stosowany w programowaniu, szczególnie w kontekście tworzenia aplikacji z interfejsem użytkownika. Celem **ViewModel** jest oddzielenie logiki biznesowej aplikacji od jej warstwy prezentacji.



```
class WordViewModel : ViewModel() {  
    private var _wordsList: MutableList<String> = mutableListOf()  
    val wordList: List<String>  
        get() = _wordsList  
  
    init {  
        reinitialize()  
    }  
  
    fun addWord(word: String){  
        _wordsList.add(word)  
        _wordsList.sort()  
    }  
  
    fun reinitialize(){  
        _wordsList.clear()  
        _wordsList.addAll(DataProvider.words)  
        _wordsList.sort()  
    }  
  
    fun clear(){  
        _wordsList.clear()  
    }  
}
```



```
class WordViewModel : ViewModel() {  
    private var _wordsList = mutableStateListOf<String>()  
    val wordList: List<String>  
        get() = _wordsList  
  
    init {  
        reinitialize()  
    }  
  
    fun addWord(word: String){  
        _wordsList.add(word)  
        _wordsList.sort()  
    }  
  
    fun reinitialize(){  
        _wordsList.clear()  
        _wordsList.addAll(DataProvider.words)  
        _wordsList.sort()  
    }  
  
    fun clear(){  
        _wordsList.clear()  
    }  
}
```

UWAGA

W aplikacjach opartych na języku **Kotlin** częściej wykorzystuje się `Flow`, `StateFlow`, lub `SharedFlow` (które poznamy na kolejnych zajęciach). Również można tworzyć aplikacje oparte na `LiveData` lecz jest to rzadziej spotykane.



```

class MyViewModel(myRepository: MyRepository) : ViewModel() {
    val data : LiveData<String> = myRepository.data
        .onStart { "Loading" }
        .map { "data converted for the view" }
        .asLiveData()
}
  
```

```

class MyRepository() {
    val data: Flow<String> = flow { this: FlowCollector<String>
        emit( value: "networkDataModel")
    }
}
  
```

```

class MyFragment() : Fragment() {

    private val myViewModel : MyViewModel by viewModels<MyViewModel>()

    private val dataObserver = Observer<String> { it: String!
        // "Loading"
        // "data converted for the view"
    }

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)

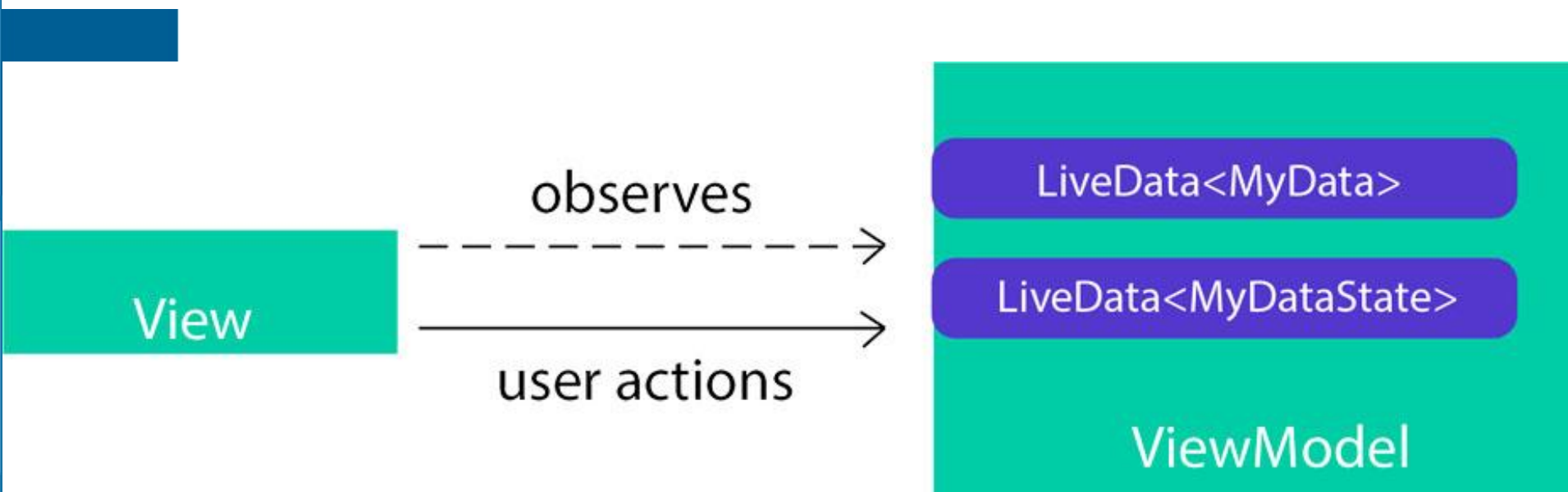
        observe(myViewModel.data, dataObserver)
    }
}

// Presentation layer extension
private fun <T> LifecycleOwner.observe(liveData: LiveData<T>, observer: Observer<T>) {
    liveData.observe( owner: this, observer)
}
  
```


UWAGA

W aplikacjach opartych na języku **Kotlin** częściej wykorzystuje się `Flow`, `StateFlow`, lub `SharedFlow` (które poznamy na kolejnych zajęciach). Również można tworzyć aplikacje oparte na `LiveData` lecz jest to rzadziej spotykane.

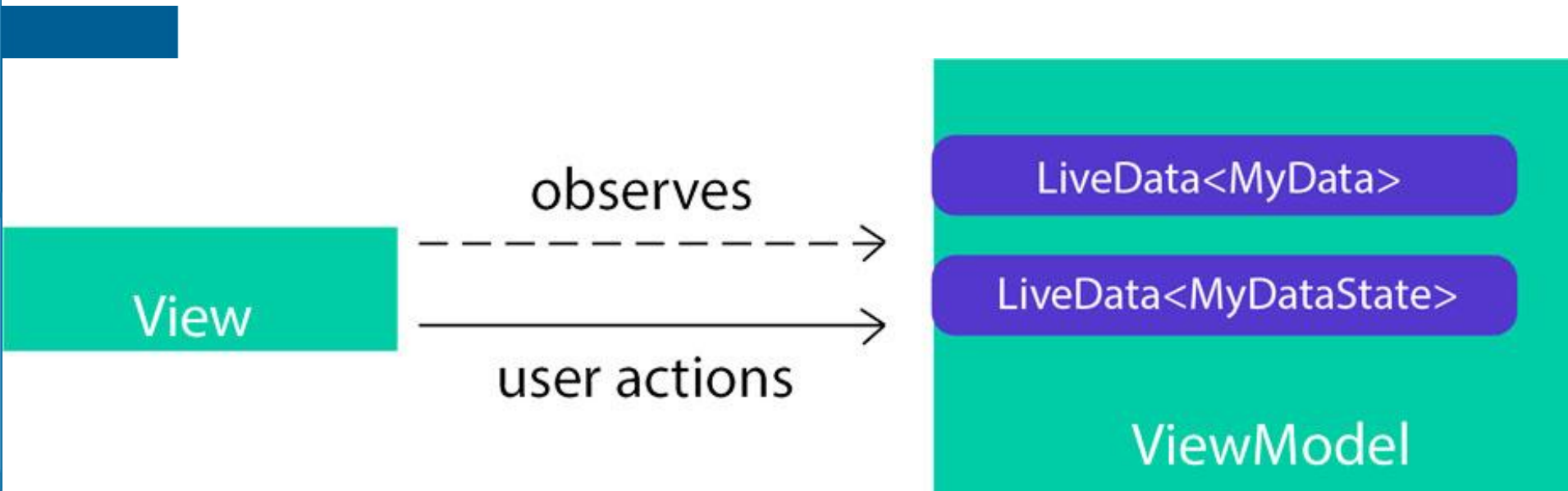
`LiveData` jest częścią bibliotek Androida i jest obiektem, który przechowuje dane **obserwowane** przez komponenty aplikacji, takie jak aktywności, fragmenty czy `ViewModel`. Jest zaprojektowany tak, aby dostarczać reaktywne i cykliczne powiadomienia o zmianach danych.



UWAGA

W aplikacjach opartych na języku **Kotlin** częściej wykorzystuje się `Flow`, `StateFlow`, lub `SharedFlow` (które poznamy na kolejnych zajęciach). Również można tworzyć aplikacje oparte na `LiveData` lecz jest to rzadziej spotykane.

`LiveData` jest częścią bibliotek Androida i jest obiektem, który przechowuje dane **obserwowane** przez komponenty aplikacji, takie jak aktywności, fragmenty czy `ViewModel`. Jest zaprojektowany tak, aby dostarczać reaktywne i cykliczne powiadomienia o zmianach danych.

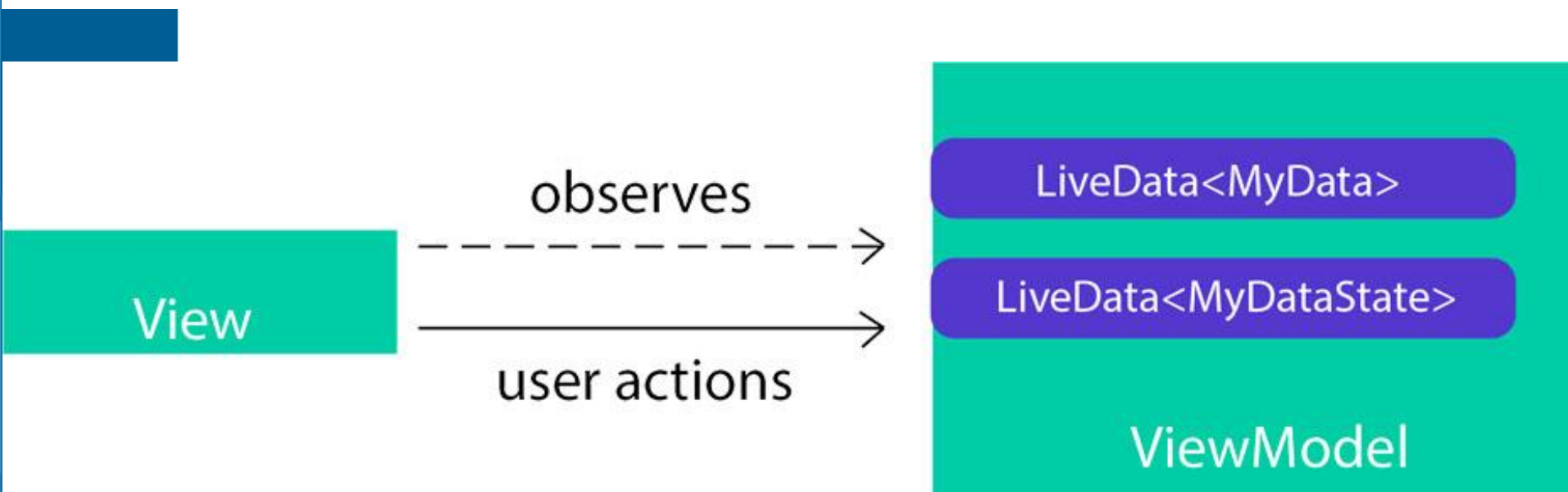


Reaktywność to podejście w programowaniu, które koncentruje się na tym, aby system reagował na zmiany i propagował te zmiany w sposób automatyczny. W kontekście aplikacji, reaktywność odnosi się do zdolności systemu do dynamicznego reagowania na zmiany danych i propagowania tych zmian do odpowiednich komponentów.

UWAGA

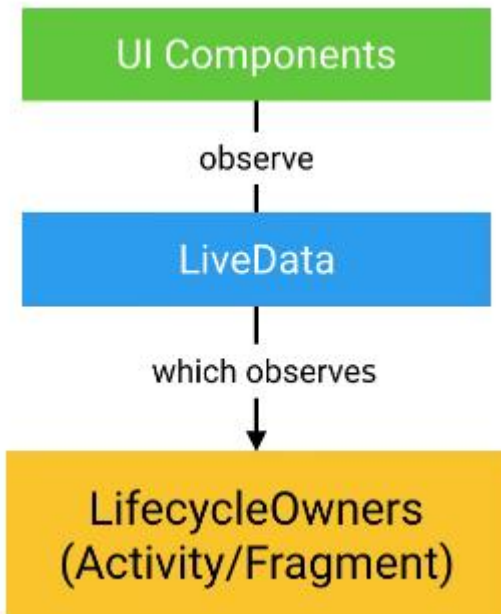
W aplikacjach opartych na języku **Kotlin** częściej wykorzystuje się `Flow`, `StateFlow`, lub `SharedFlow` (które poznamy na kolejnych zajęciach). Również można tworzyć aplikacje oparte na `LiveData` lecz jest to rzadziej spotykane.

`LiveData` jest częścią bibliotek Androida i jest obiektem, który przechowuje dane **obserwowane** przez komponenty aplikacji, takie jak aktywności, fragmenty czy `ViewModel`. Jest zaprojektowany tak, aby dostarczać reaktywne i cykliczne powiadomienia o zmianach danych.



Reaktywność to podejście w programowaniu, które koncentruje się na tym, aby system reagował na zmiany i propagował te zmiany w sposób automatyczny. W kontekście aplikacji, reaktywność odnosi się do zdolności systemu do dynamicznego reagowania na zmiany danych i propagowania tych zmian do odpowiednich komponentów.

Opiera się na obsłudze zdarzeń, które są generowane w systemie w wyniku **zmian w danych**. Zdarzenia te mogą być przekształcane i łączone za pomocą różnych operacji, tworząc **strumienie danych**. Strumienie są sekwencją wartości, które mogą być emitowane i subskrybowane (obserwowane) przez komponenty. Automatycznie propaguje zmiany w danych do komponentów, które subskrybują te dane. Oznacza to, że komponenty **nie muszą** ręcznie monitorować i aktualizować danych, ponieważ system sam zarządza tym procesem.



1. Zapewnia, że stan interfejsu użytkownika są zgodny z aktualnym stanem danych. LiveData powiadamia obiekty Observer, gdy zmienia się stan cyklu życia.

2. Brak wycieków pamięci. Obserwatorzy są związani z obiektami cyklu życia.

3. Brak błędów z powodu zatrzymanych aktywności. Jeśli cykl życia obserwatora jest nieaktywny, nie otrzymuje on żadnych zdarzeń LiveData.

4. Komponenty interfejsu użytkownika jedynie obserwują istotne dane i nie przerywają ani nie wznowiają obserwacji. LiveData automatycznie zarządza wszystkim tym, ponieważ jest „świadoma” istotnych zmian stanu cyklu życia podczas obserwacji.

5. Zawsze aktualne dane. Jeśli cykl życia staje się nieaktywny, otrzymuje najnowsze dane po ponownym staniu się aktywnym. Na przykład aktywność, która była w tle, otrzymuje najnowsze dane zaraz po powrocie do pierwszego planu.

```
class CounterViewModel : ViewModel() {  
    private var _counter = MutableLiveData(0)  
    val counter: LiveData<Int>  
        get() = _counter  
  
    fun increase(){  
        _counter.value = _counter.value?.inc()  
    }  
  
    fun decrease(){  
        _counter.value = _counter.value?.dec()  
    }  
  
    fun clear(){ _counter.value = 0 }  
}
```

UWAGA

W aplikacjach opartych na **JetpackCompose** wykorzystuje się **State** (który widzieliśmy w poprzednim przykładzie), **StateFlow**, lub **ComposeFlow** (które poznamy na kolejnych zajęciach). Również można tworzyć aplikacje oparte na **LiveData** lecz jest to rzadziej spotykane. W praktyce posiadając **LiveData** w aplikacji konwertujemy go do **State** przez metodę **observeAsState**, z którą zapoznamy się w tym przykładzie.

```
@Composable
fun CounterScreen() {

    val viewModel: CounterViewModel = viewModel() // zapewnia dostęp do metod dostępowych i danych (tylko do
    val counterState = viewModel.counter.observeAsState() // przy każdej zmianie wartości counter,
                                                         // counterState otrzymuje aktualną wartość
```