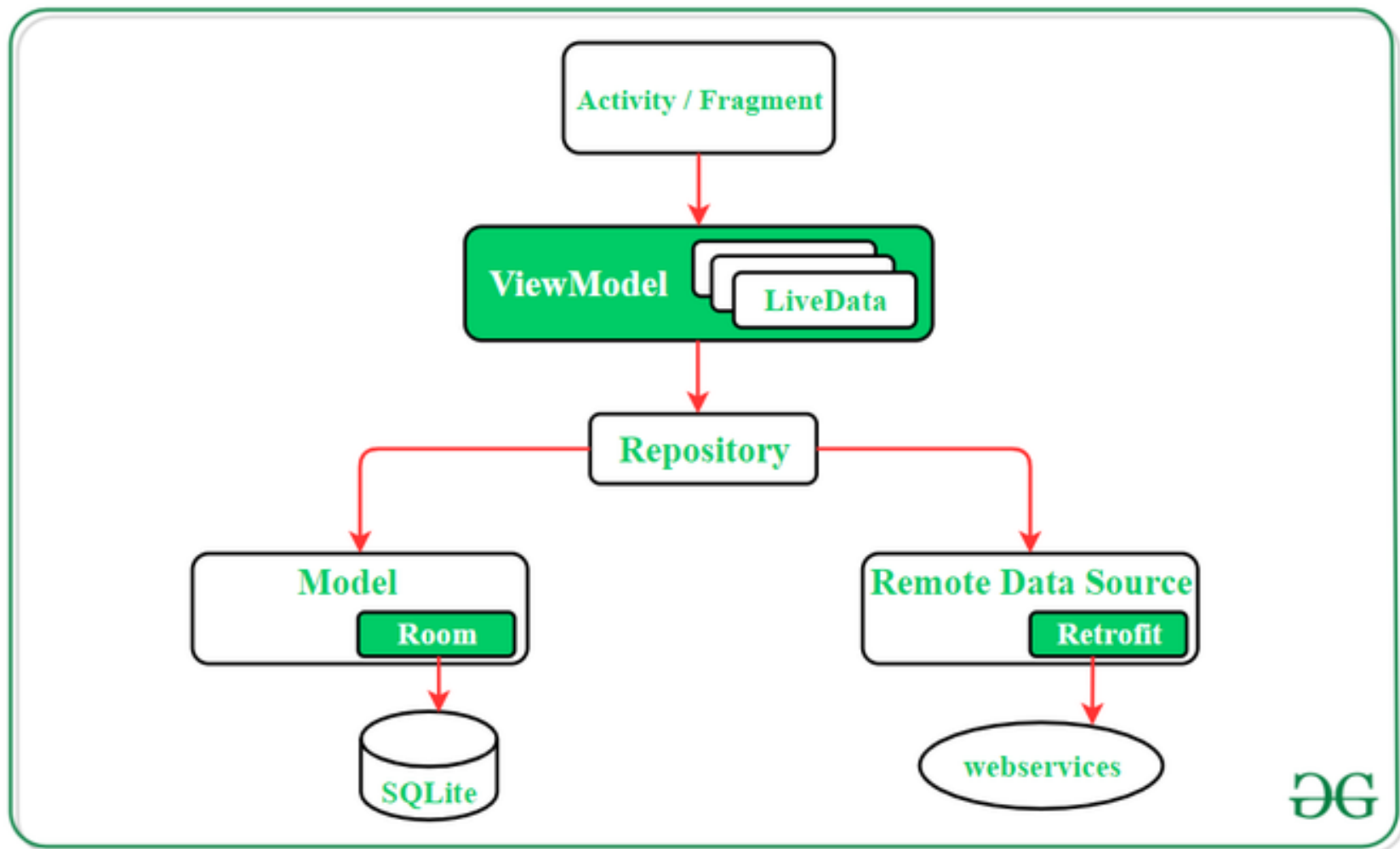




PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

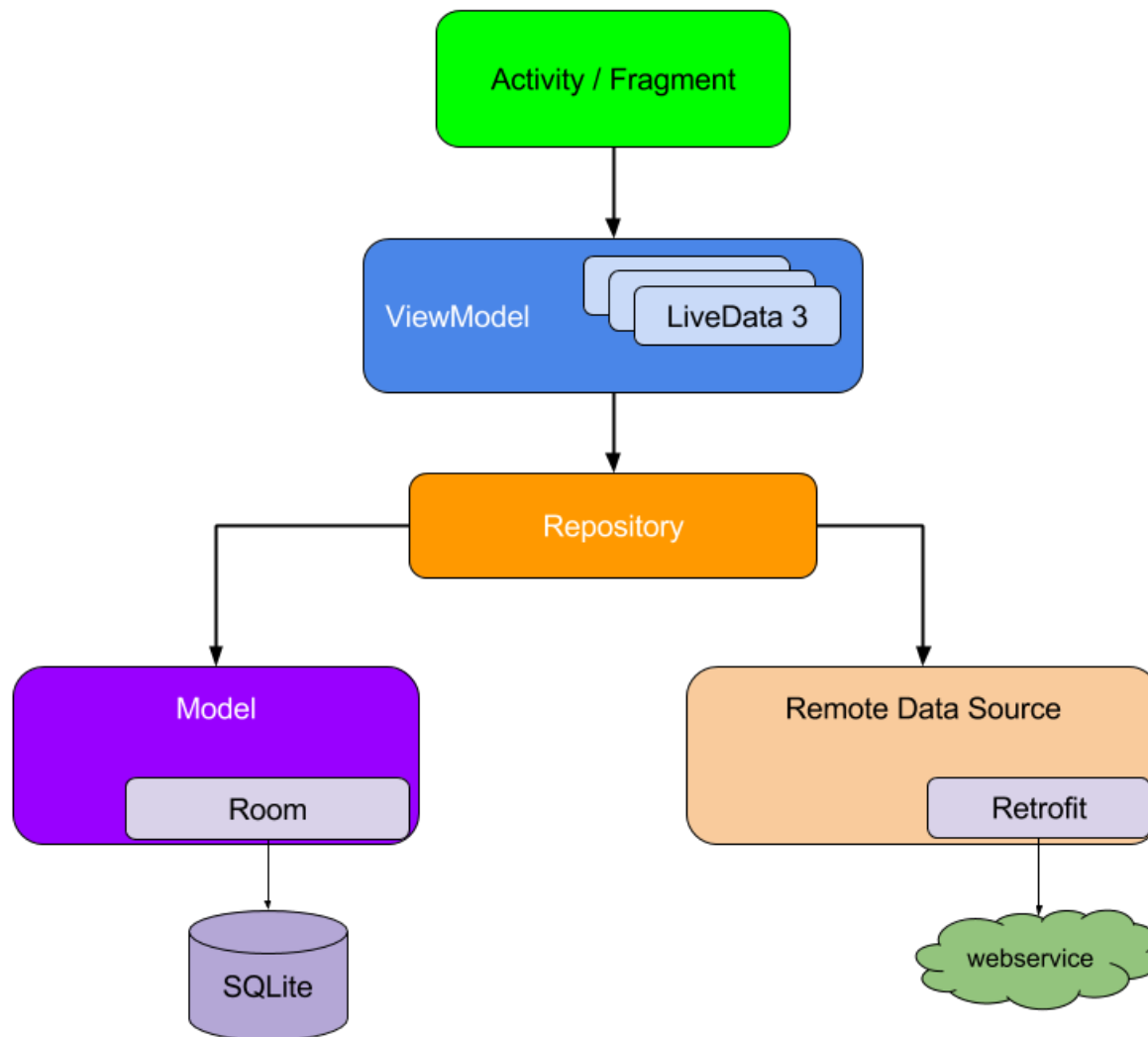
WYKŁAD 8 Architektura Aplikacji 1

- ViewModel
- LiveData
- MVVM

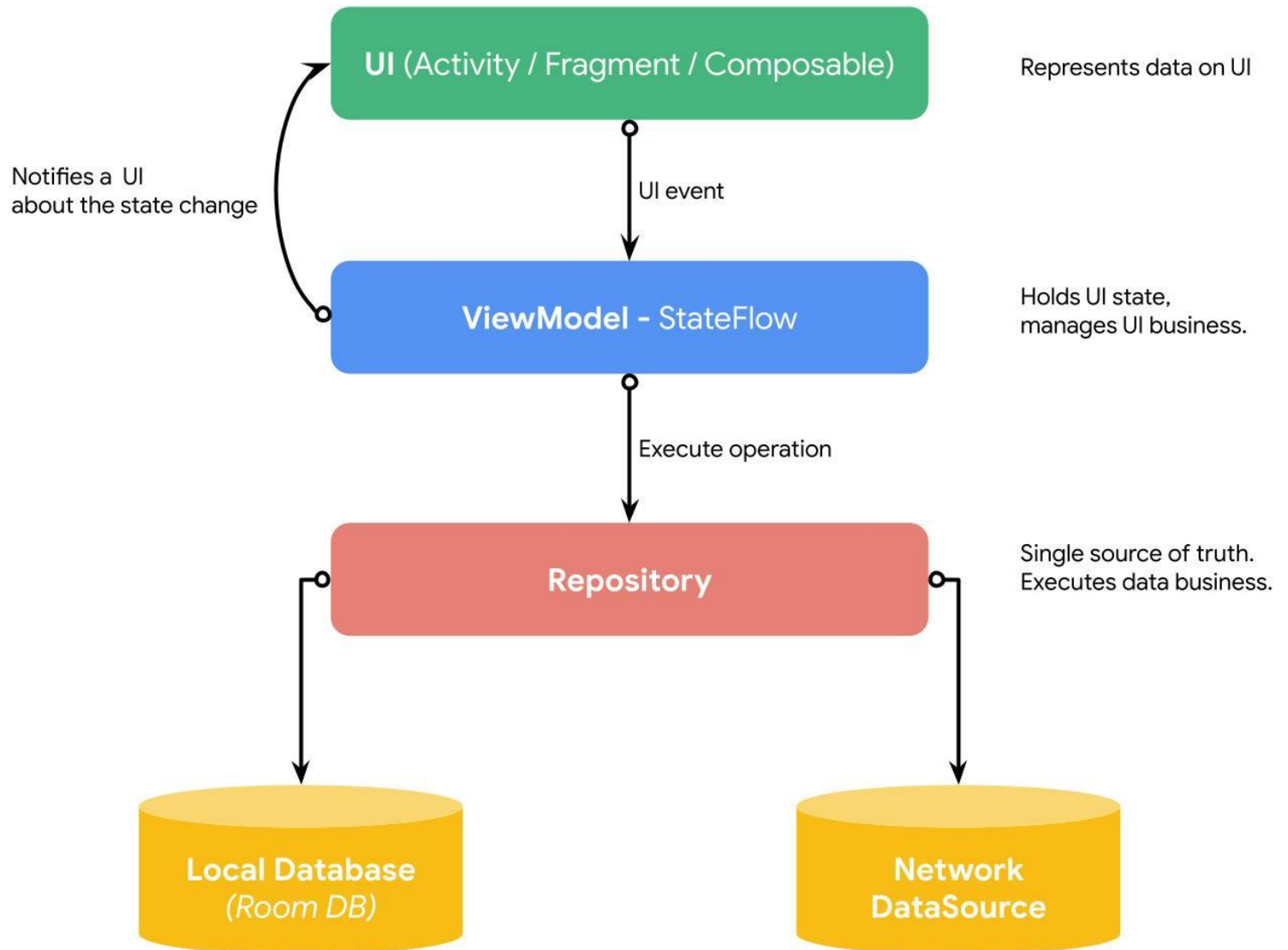


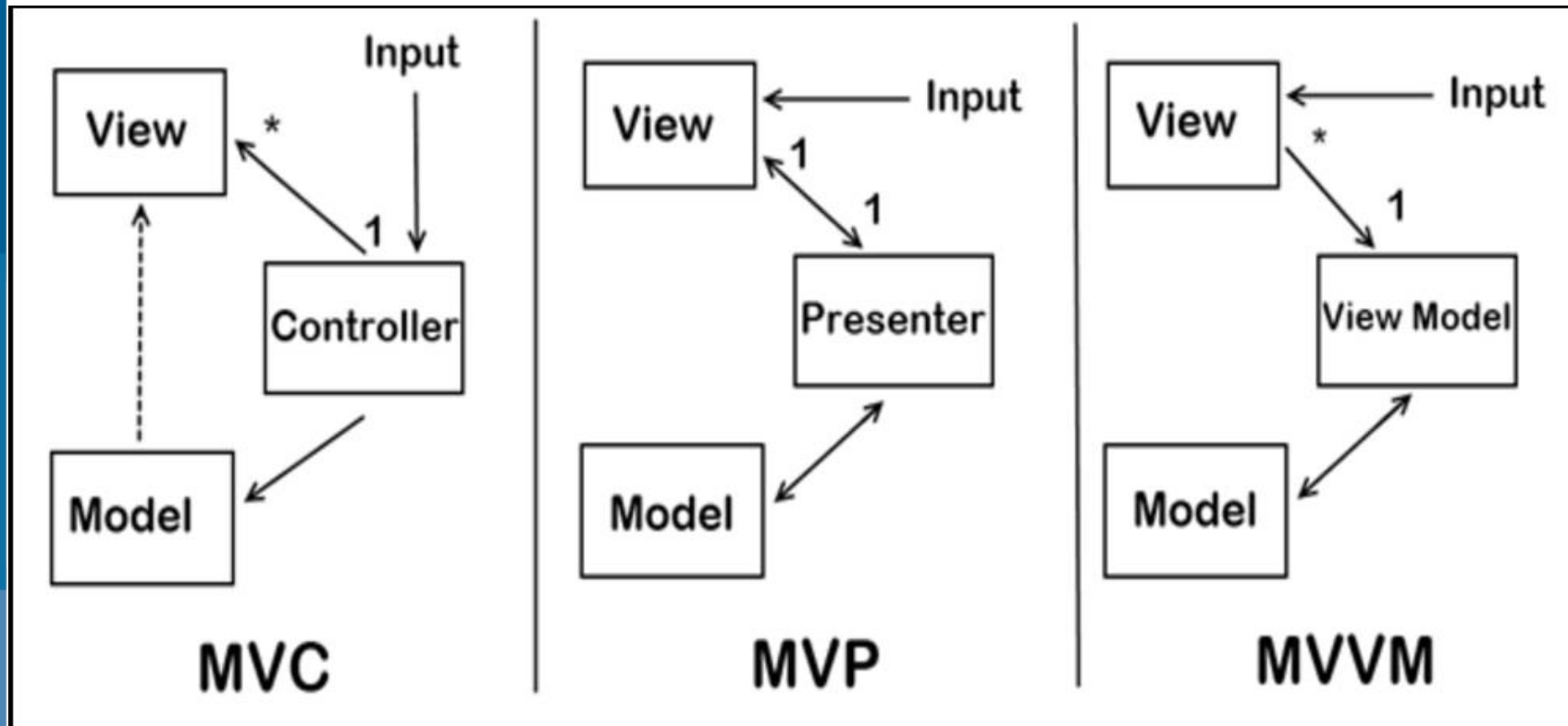
`ViewModel` to wzorzec projektowy stosowany w programowaniu, szczególnie w kontekście tworzenia aplikacji z interfejsem użytkownika. Celem `ViewModel` jest oddzielenie logiki biznesowej aplikacji od jej warstwy prezentacji.

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.

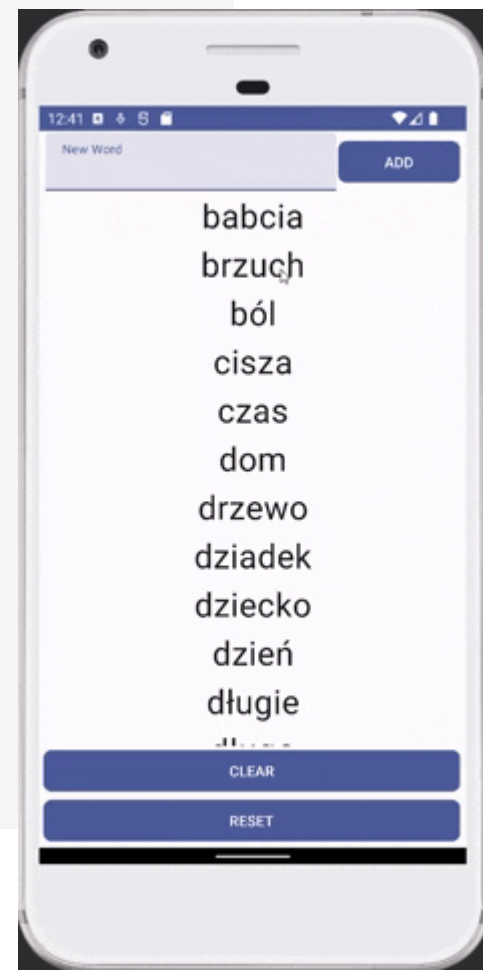


ViewModel

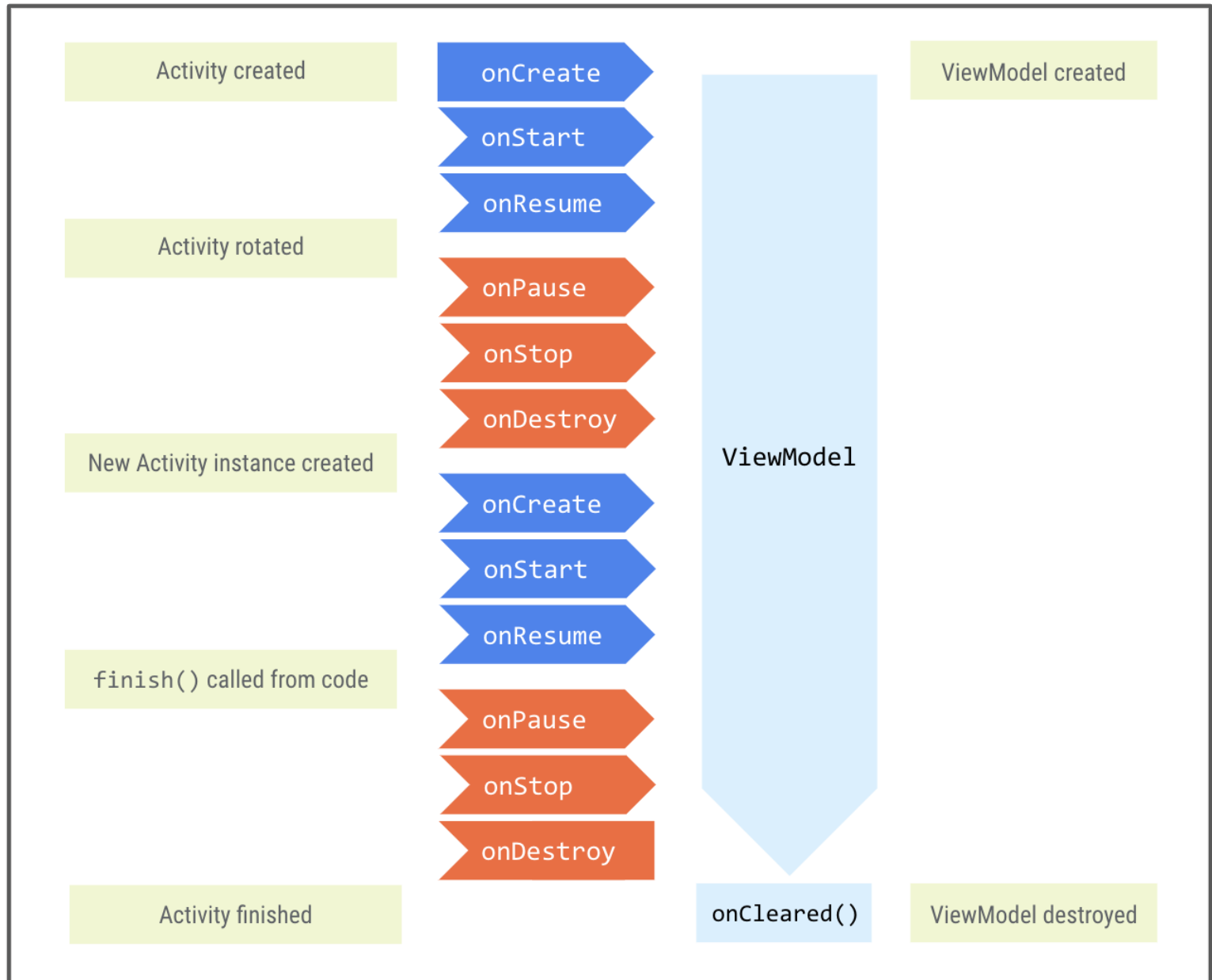




```
class WordViewModel : ViewModel() {  
    private var _wordsList: MutableList<String> = mutableListOf()  
    val wordList: List<String>  
        get() = _wordsList  
  
    init {  
        reinitialize()  
    }  
  
    fun addWord(word: String){  
        _wordsList.add(word)  
        _wordsList.sort()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



ViewModel



RecyclerView

```
class ViewHolder(private val binding: RvItemBinding) : RecyclerView.ViewHolder(binding.root) {  
    fun bind(item: String) {  
        binding.wordTextView.text = item  
    }  
}
```

```
class WordViewHolder(private val binding: RvItemBinding) : RecyclerView.ViewHolder(binding.root) {  
    fun bind(item: String) {  
        binding.wordTextView.text = item  
    }  
}
```

```
class WordComparator : DiffUtil.ItemCallback<String>() {  
    override fun areItemsTheSame(oldItem: String, newItem: String): Boolean {  
        return oldItem === newItem  
    }  
  
    override fun areContentsTheSame(oldItem: String, newItem: String): Boolean {  
        return oldItem == newItem  
    }  
}
```

```
class ViewHolder(private val binding: RvItemBinding) : RecyclerView.ViewHolder(binding.root) {  
    fun bind(item: String) {  
        binding.wordTextView.text = item  
    }  
}
```

```
class WordComparator : DiffUtil.ItemCallback<String>() {  
    override fun areItemsTheSame(oldItem: String, newItem: String): Boolean {  
        return oldItem === newItem  
    }  
  
    override fun areContentsTheSame(oldItem: String, newItem: String): Boolean {  
        return oldItem == newItem  
    }  
}
```

```
class WordAdapter(wordComparator: WordComparator) : ListAdapter<String, ViewHolder>(wordComparator) {  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): ViewHolder {  
        return ViewHolder(  
            RvItemBinding.inflate(  
                LayoutInflater.from(parent.context), parent, false  
            )  
        )  
    }  
  
    override fun onBindViewHolder(holder: ViewHolder, position: Int) {  
        val item = getItem(position)  
        holder.bind(item)  
    }  
}
```

```
private fun onAddWord() {  
    val word = binding.wordEditText.text.toString()  
    viewModel.addWord(word)  
    wordAdapter.notifyDataSetChanged()  
}
```

```
private fun onResetWords(){  
    viewModel.reinitialize()  
    wordAdapter.notifyDataSetChanged()  
}
```

```
private fun onClearWords(){  
    viewModel.clear()  
    wordAdapter.notifyDataSetChanged()  
}
```

```
override fun onCreateView(  
    inflater: LayoutInflater, container: ViewGroup?,  
    savedInstanceState: Bundle?  
): View {  
    binding = FragmentListBinding.inflate(inflater)  
  
    wordAdapter.submitList(viewModel.wordList)  
  
    binding.rvList.apply{  
        adapter = wordAdapter  
        layoutManager = LinearLayoutManager(requireContext())  
    }  
  
    binding.addButton.setOnClickListener { onAddWord() }  
    binding.resetButton.setOnClickListener { onResetWords() }  
    binding.clearButton.setOnClickListener { onClearWords() }  
  
    return binding.root  
}
```

```
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()  
    }  
}
```

Compose

```
LazyColumn(modifier = Modifier.weight(1f)){
    items(viewModel.wordList.size){
        Text(
            text = viewModel.wordList[it],
            fontSize = 32.sp,
            textAlign = TextAlign.Center,
            modifier = Modifier
                .fillMaxWidth()
                .padding(2.dp)
        )
    }
}
```

```
class WordViewModel : ViewModel() {
    private var _wordList = mutableStateListOf<String>()
    val wordList: List<String>
        get() = _wordList

    init {
        reinitialize()
    }

    fun addWord(word: String){
        _wordList.add(word)
        _wordList.sort()
    }

    fun reinitialize(){
        _wordList.clear()
        _wordList.addAll(DataProvider.words)
        _wordList.sort()
    }

    fun clear(){
        _wordList.clear()
    }
}
```

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ListScreen(){

    var word by remember { mutableStateOf("") }
    val viewModel: WordViewModel = viewModel()
```

Compose

```
@OptIn(ExperimentalMaterial3Api::class)
@Composable
fun ListScreen(){

    var word by remember { mutableStateOf("") }
    val viewModel: WordViewModel = viewModel()
```

```
Button(
    modifier = Modifier
        .weight(1f)
        .padding(start = 2.dp, end = 4.dp),
    shape = RoundedCornerShape(8.dp),
    onClick = {
        if (word.isNotEmpty()) {
            viewModel.addWord(word)
        }
    }
) {
    Text(text = "ADD")
}
```

```
Button(
    modifier = Modifier
        .fillMaxWidth()
        .padding(start = 2.dp, end = 4.dp),
    shape = RoundedCornerShape(8.dp),
    onClick = { viewModel.clear() }
) {
    Text(text = "CLEAR")
}

Button(
    modifier = Modifier
        .fillMaxWidth()
        .padding(start = 2.dp, end = 4.dp),
    shape = RoundedCornerShape(8.dp),
    onClick = { viewModel.reinitialize() }
) {
    Text(text = "RESET")
}
```

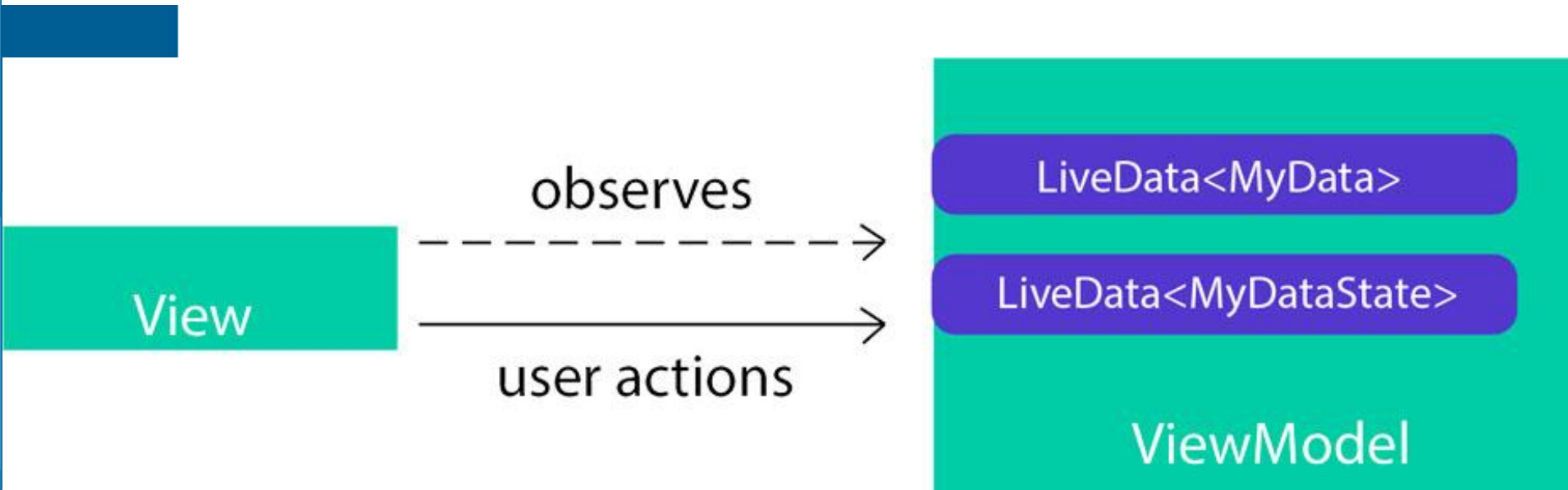
UWAGA

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.

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.

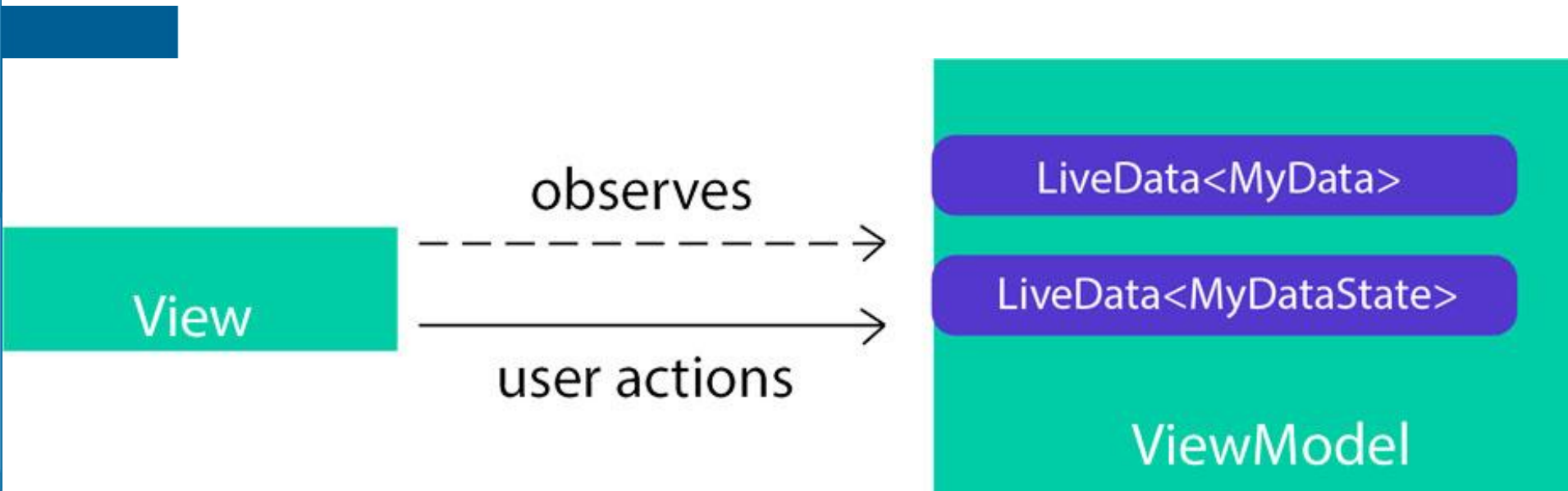
`LiveData` jest częścią bibliotek Androida i jest obiektem, który przechowuje dane **obserwowane** przez komponenty aplikacji, takie jak aktywności, fragmenty czy `ViewModel`. Jest zaprojektowany tak, aby dostarczać reaktywne i cykliczne powiadomienia o zmianach danych.



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.

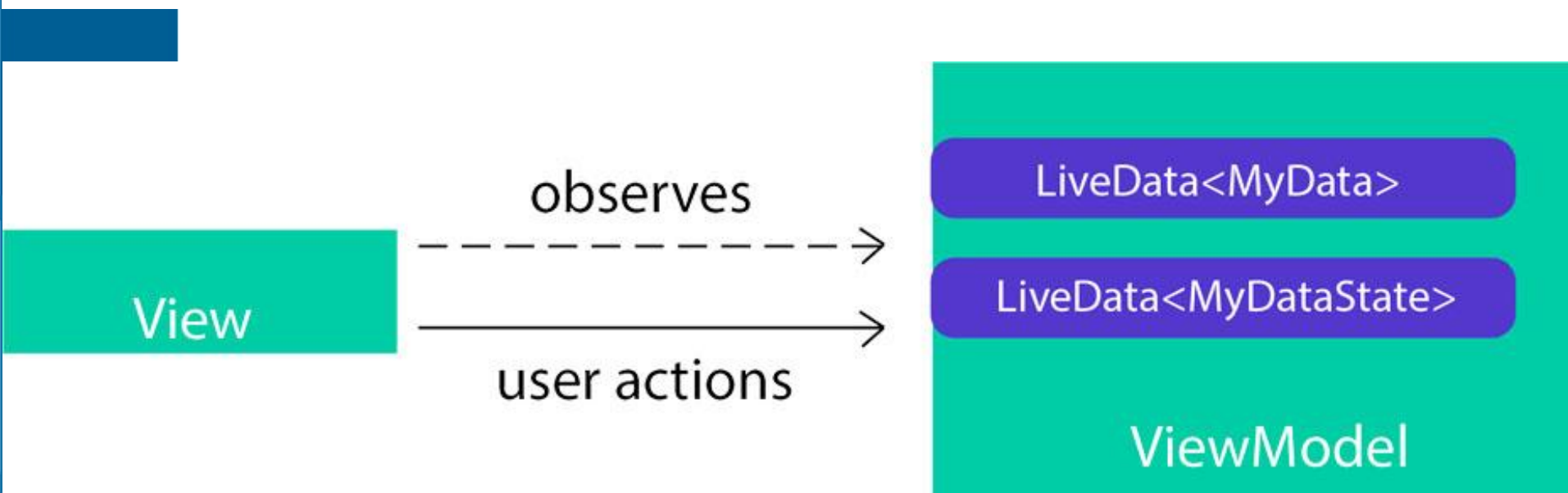


Reaktywność to podejście w programowaniu, które koncentruje się na tym, aby system reagował na zmiany i propagował te zmiany w sposób automatyczny. W kontekście aplikacji, reaktywność odnosi się do zdolności systemu do dynamicznego reagowania na zmiany danych i propagowania tych zmian do odpowiednich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

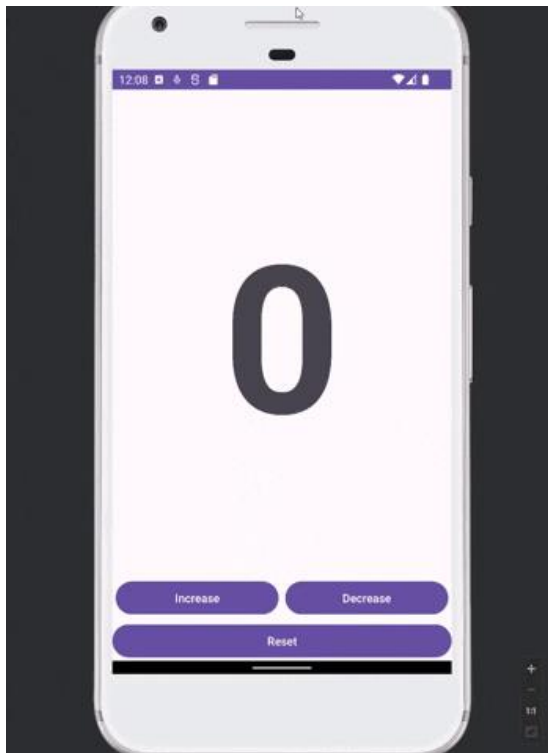
Opiera się na obsłudze zdarzeń, które są generowane w systemie w wyniku **zmian w danych**. Zdarzenia te mogą być przekształcane i łączone za pomocą różnych operacji, tworząc **strumienie danych**. Strumienie są sekwencją wartości, które mogą być emitowane i subskrybowane (obserwowane) przez komponenty. 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.

```
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 }  
}
```

```
private val viewModel: CounterViewModel by viewModels()
```

```
viewModel.counter.observe(viewLifecycleOwner) { newValue ->  
    binding.showCount.text = newValue.toString()  
}
```

```
binding.increaseButton.setOnClickListener { viewModel.increase() }  
binding.decreaseButton.setOnClickListener { viewModel.decrease() }  
binding.resetButton.setOnClickListener { viewModel.clear() }
```



```
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
```