Uniwersytet Wrocławski

Rafał Lewandków

PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 2

WYKŁAD 4

- Podstawy Architektury Aplikacji
- Wzorce MVVM

Presentation Layer Architectural Pattern

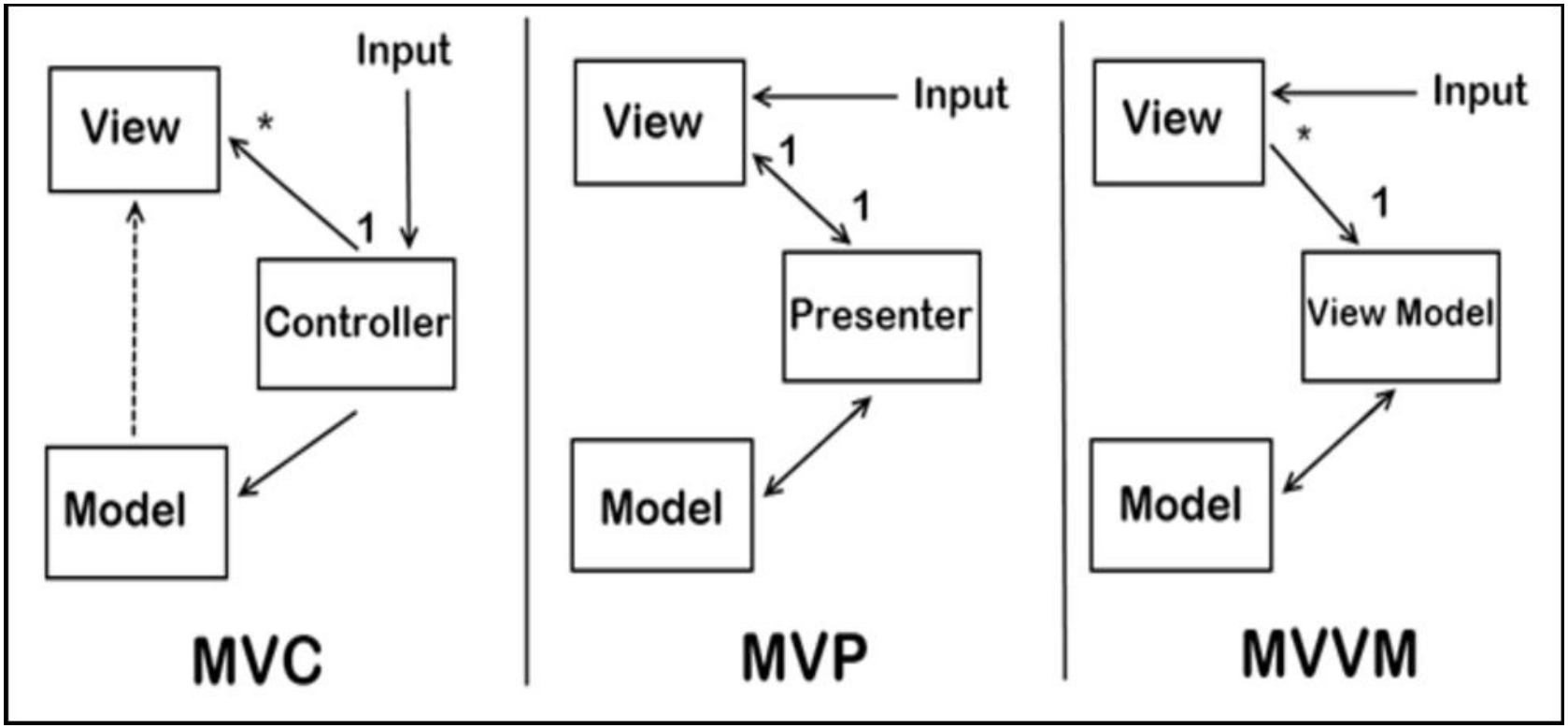
Wysokopoziomowa struktura wielokrotnego użytku przeznaczona do organizacji systemu.

Presentation Layer Architectural Pattern

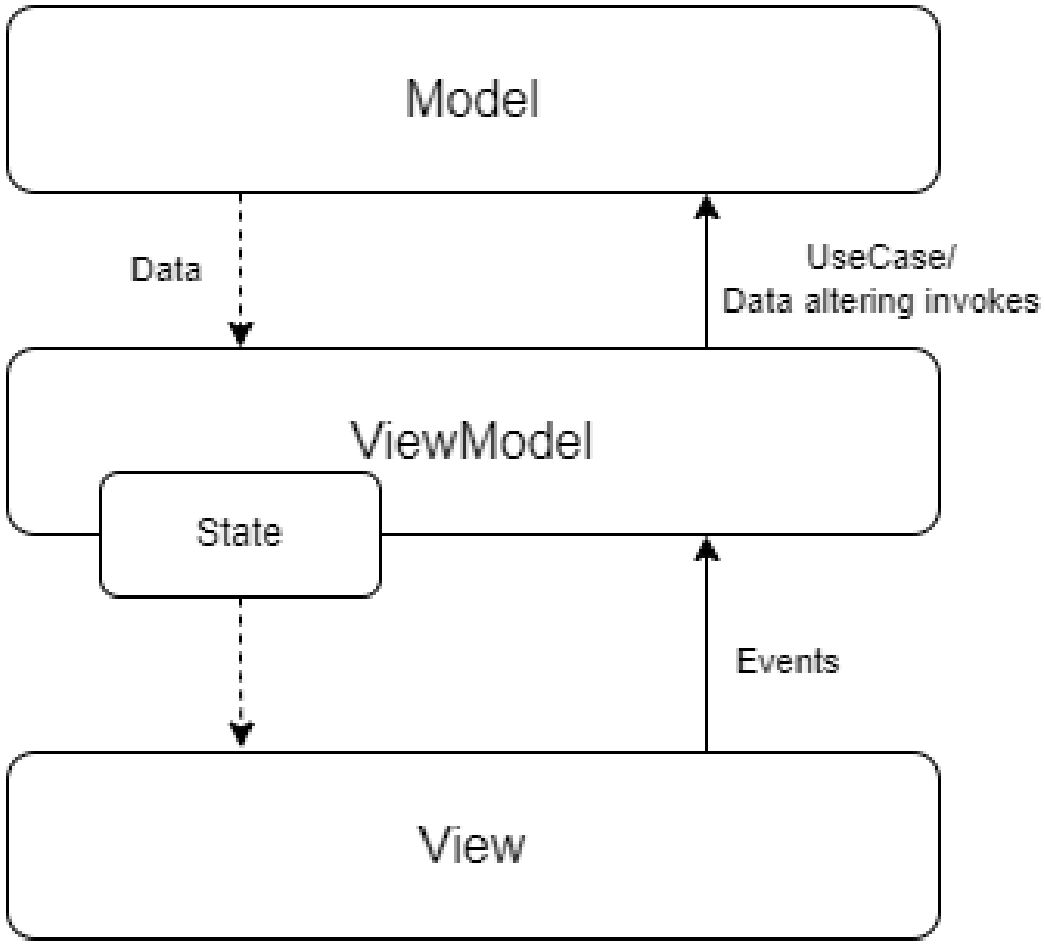
Wysokopoziomowa struktura wielokrotnego użytku przeznaczona do organizacji systemu.

- **Prezentacja danych:**
  - Wyświetla dane, które są dostarczane przez warstwę domeny.
  - Odpowiada za formatowanie i renderowanie danych w odpowiedniej formie, aby użytkownik mógł je zrozumieć i nimi manipulować.
- **Obsługa interakcji użytkownika:**
  - Reaguje na akcje użytkownika, takie jak kliknięcia przycisków, wpisywanie tekstu, przewijanie, itp.
  - Przekazuje informacje o akcjach użytkownika do warstwy domeny lub bezpośrednio do modelu danych w celu zaktualizowania stanu aplikacji.
- **Zarządzanie cyklem życia komponentów UI:**
  - Zajmuje się zarządzaniem stanem i cyklem życia komponentów UI, takich jak Activity, Fragment, ViewModel (w architekturze MVVM) oraz innych elementów UI.
  - Utrzymuje spójność UI podczas zmian, takich jak rotacja ekranu, zamykanie i ponowne otwieranie aplikacji.

MVx


View
Input
*
1
Controller
Model
MVC
View
Input
1
1
Presenter
Model
MVP
View
Input
*
1
View Model
Model
MVVM

MVx

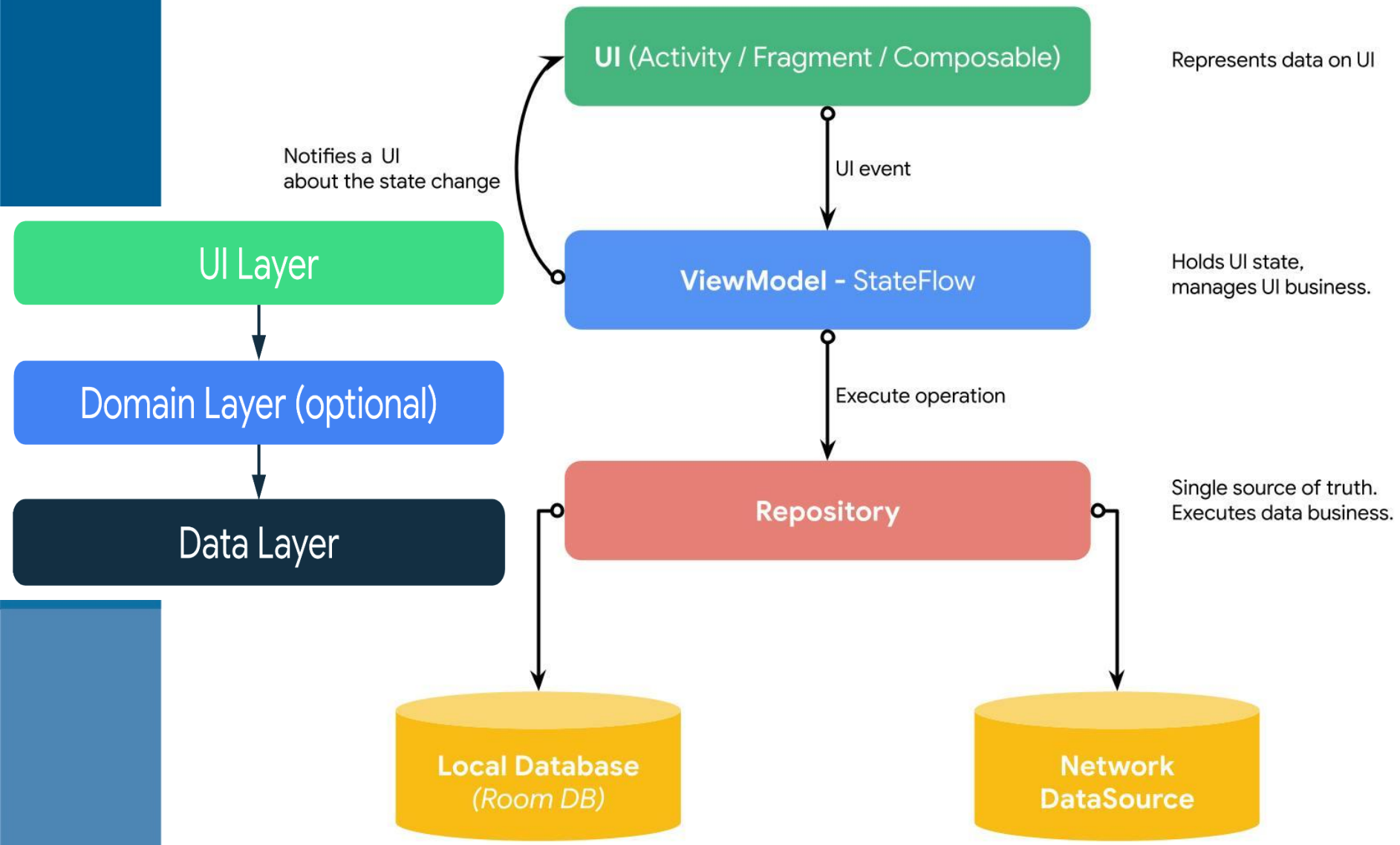


- Repositories
- DataSources
- Business Logic
- Network calls
- Creating, storing and changing app data

- Interactors logic
- SSOT (Single Source Of Truth)
- UDF (Unidirectional Data Flow)
- Only expose the State

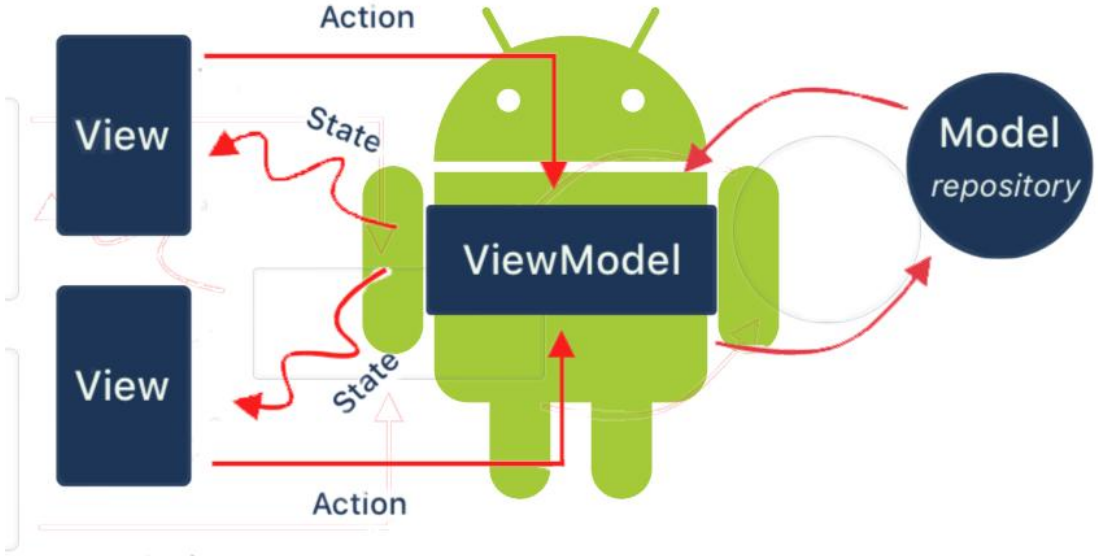
- User Interface
- Animations
- Context
- Buttons, Texts etc.

MVVM


UI (Activity / Fragment / Composable)
Represents data on UI
UI event
Notifies a UI
about the state change
UI Layer
ViewModel - StateFlow
Holds UI state,
manages UI business.
Domain Layer (optional)
Execute operation
Repository
Single source of truth.
Executes data business.
Data Layer
Local Database
(Room DB)
Network
DataSource

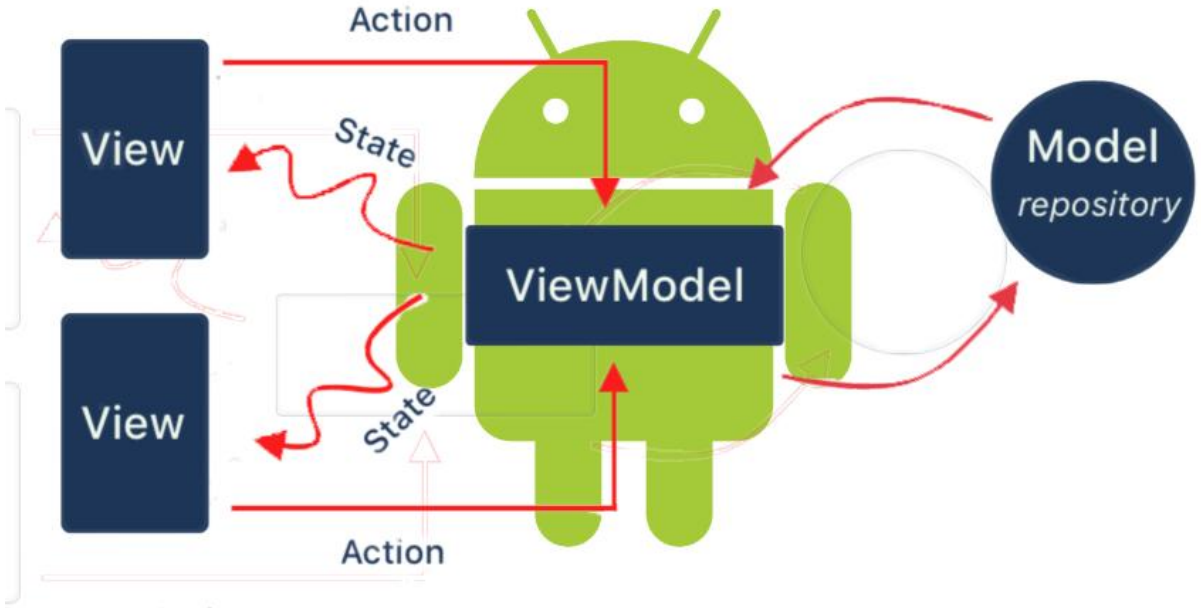
MVVM

- **View** (Widok - Kelner): Nasz komponent **Composable**. Jest "*głupi*" – nie podejmuje decyzji. Jego zadania to:
  - Wyświetlić dane, które otrzymał (State).
  - Poinformować o akcji użytkownika (Event), np. "klient wcisnął przycisk 'Zapisz'".
- **ViewModel** (Szef Kuchni): To mózg operacji dla danego ekranu. Przechowuje stan UI (np. czy widoczny jest wskaźnik ładowania, jaka jest zawartość pola tekstowego).Zawiera logikę biznesową (np. co się dzieje po kliknięciu "Zapisz"). Komunikuje się z Modelem, aby pobrać lub zapisać dane. Nie ma żadnej wiedzy o frameworku UI Androida. Nie wie, czym jest Composable czy Button.
- **Model** (Magazyn i Dostawcy): Warstwa danych. Repozytoria, źródła danych (API, baza danych).Odpowiada za dostarczanie i zapisywanie danych.

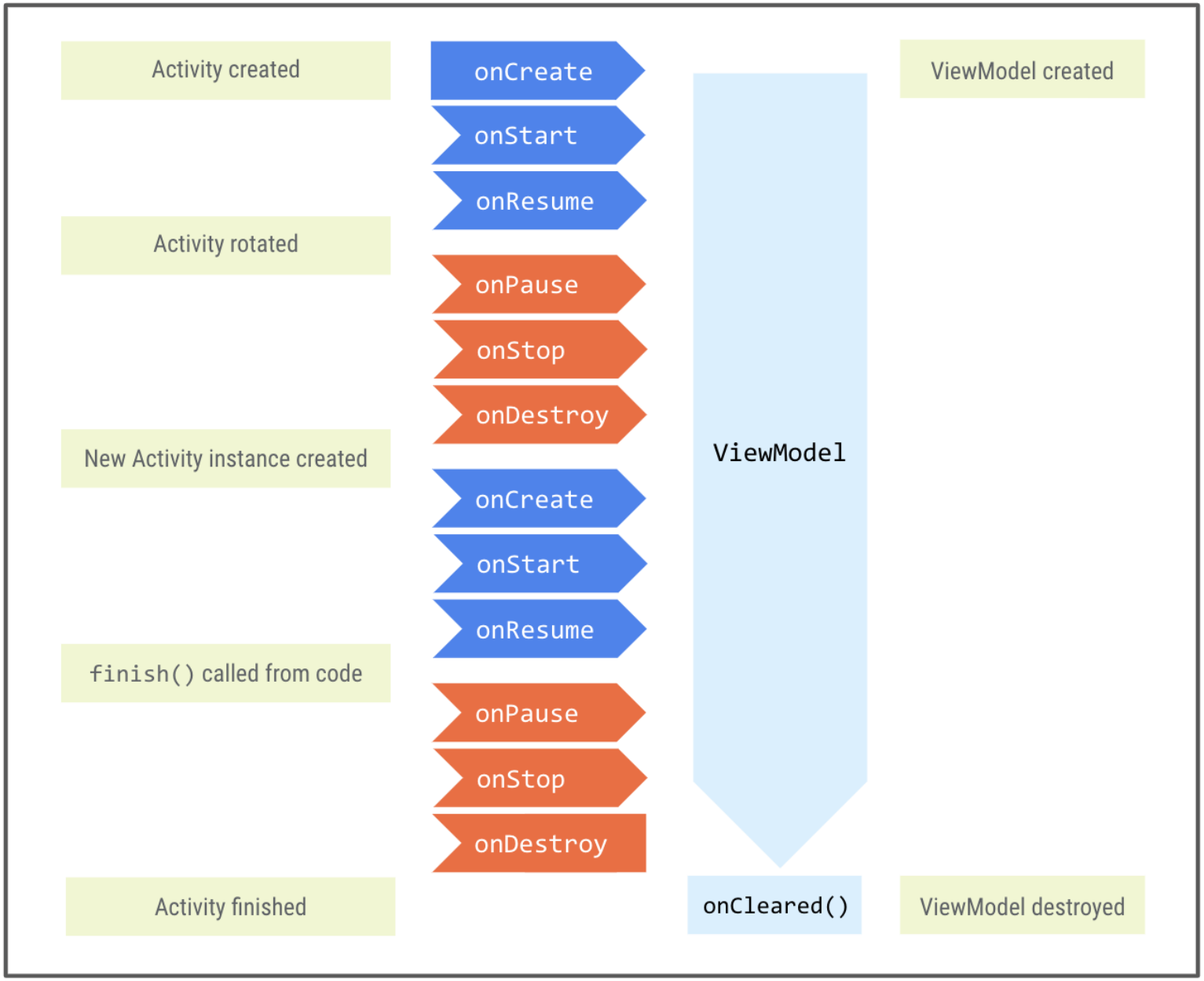


MVVM

- **View -> ViewModel**: Użytkownik wykonuje akcję (np. klika przycisk), a View informuje o tym ViewModel, wywołując jego funkcję.
- **ViewModel -> Model**: ViewModel przetwarza akcję i prosi Model o dane.Model -> ViewModel: Model zwraca dane do ViewModel.
- **ViewModel -> View**: ViewModel aktualizuje swój wewnętrzny stan, a View, który ten stan obserwuje, automatycznie odświeża interfejs, aby odzwierciedlić nowe dane.

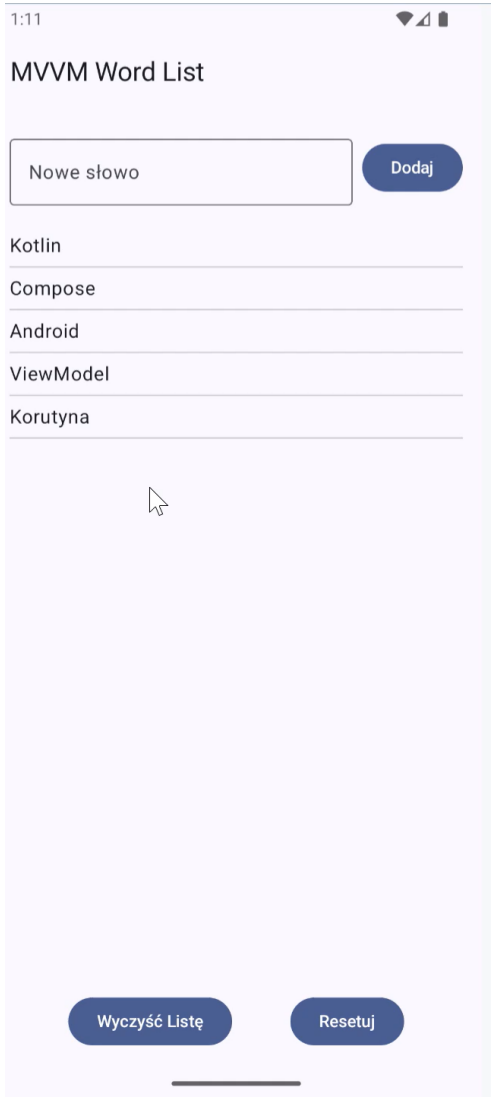


MVVM


Activity created
onCreate
onStart
onResume
ViewModel created
Activity rotated
onPause
onStop
onDestroy
New Activity instance created
ViewModel
onCreate
onStart
onResume
finish() called from code
onPause
onStop
onDestroy
Activity finished
onCleared()
ViewModel destroyed

ViewModel

Do tej pory trzymaliśmy dane razem z elementacmi @Composable, dzięki MVVM **rozdzielimy** odpowiedzialności. Composable zajmuje się **tylko wyświetlaniem**. Cała **logika i stan** znajdują się w ViewModelu.



ViewModel

Dziedziczenie po klasie ViewModel daje dostęp do metod **kluczowych** dla architektury aplikacji. Między innymi: **przeżywanie zmian konfiguracji**, **świadomość cyklu życia** (ViewModel posiada wbudowany **viewModelScope**)

```
class WordViewModel : ViewModel() {

    private val initialWords = listOf("Kotlin", "Compose",
    val words = mutableStateListOf<String>()
    var textFieldValue by mutableStateOf( value = "")
        private set

    init {
        resetWords()
    }
    fun onTextFieldValueChanged(newValue: String) {...}

    fun addWord() {
        if (textFieldValue.isNotBlank()) {
            viewModelScope.launch {
                delay( timeMillis = 300)
                words.add(textFieldValue.trim())
                textFieldValue = ""
            }
        }
    }

    fun clearWords() {...}

    fun resetWords() {...}
}
```

ViewModel

Dziedziczenie po klasie ViewModel daje dostęp do metod **kluczowych** dla architektury aplikacji. Między innymi: **przeżywanie zmian konfiguracji**, **świadomość cyklu życia** (ViewModel posiada wbudowany **viewModelScope**)

mutableStateOf("") tworzy specjalny, **obserwowalny** przez Compose obiekt, który **przechowuje stan** (w tym przypadku tekst). Słowo kluczowe by to **delegat właściwości**.

```
class WordViewModel : ViewModel() {

    private val initialWords = listOf("Kotlin", "Compose",
    val words = mutableStateListOf<String>()
    var textFieldValue by mutableStateOf( value = "")
        private set

    init {
        resetWords()
    }
    fun onTextFieldValueChanged(newValue: String) {...}

    fun addWord() {
        if (textFieldValue.isNotBlank()) {
            viewModelScope.launch {
                delay( timeMillis = 300)
                words.add(textFieldValue.trim())
                textFieldValue = ""
            }
        }
    }

    fun clearWords() {...}

    fun resetWords() {...}
}
```

Uniwersytet Wrocławski

ViewModel

Dziedziczenie po klasie ViewModel daje dostęp do metod **kluczowych** dla architektury aplikacji. Między innymi: **przeżywanie zmian konfiguracji**, **świadomość cyklu życia** (ViewModel posiada wbudowany **viewModelScope**)

mutableStateOf("") tworzy specjalny, **obserwowalny** przez Compose obiekt, który **przechowuje stan** (w tym przypadku tekst). Słowo kluczowe by to **delegat właściwości**.

private set to **modyfikator widoczności** zastosowany **tylko do settera** tej właściwości. Oznacza to, że operacja zapisu (textFieldValue = "nowy tekst") może być wykonana **wyłącznie wewnątrz** klasy **WordViewModel**.

```
class WordViewModel : ViewModel() {

    private val initialWords = listOf("Kotlin", "Compose",
    val words = mutableStateListOf<String>()
    var textFieldValue by mutableStateOf( value = "")
        private set

    init {
        resetWords()
    }
    fun onTextFieldValueChanged(newValue: String) {...}

    fun addWord() {
        if (textFieldValue.isNotBlank()) {
            viewModelScope.launch {
                delay( timeMillis = 300)
                words.add(textFieldValue.trim())
                textFieldValue = ""
            }
        }
    }

    fun clearWords() {...}

    fun resetWords() {...}
}
```

Uniwersytet Wrocławski

ViewModel

Dziedziczenie po klasie ViewModel daje dostęp do metod **kluczowych** dla architektury aplikacji. Między innymi: **przeżywanie zmian konfiguracji**, **świadomość cyklu życia** (ViewModel posiada wbudowany **viewModelScope**)

mutableStateOf("") tworzy specjalny, **obserwowalny** przez Compose obiekt, który **przechowuje stan** (w tym przypadku tekst). Słowo kluczowe by to **delegat właściwości**.

private set to **modyfikator widoczności** zastosowany **tylko do settera** tej właściwości. Oznacza to, że operacja zapisu (textFieldValue = "nowy tekst") może być wykonana **wyłącznie wewnątrz** klasy **WordViewModel**.

viewModelScope.launch uruchamia **nową korutynę** w **specjalnym zakresie** (scope), który jest **integralną częścią ViewModel** i **automatycznie** zarządza jej **cyklem życia**.

```
class WordViewModel : ViewModel() {

    private val initialWords = listOf("Kotlin", "Compose",
    val words = mutableStateListOf<String>()
    var textFieldValue by mutableStateOf( value = "")
        private set

    init {
        resetWords()
    }
    fun onTextFieldValueChanged(newValue: String) {...}

    fun addWord() {
        if (textFieldValue.isNotBlank()) {
            viewModelScope.launch {
                delay( timeMillis = 300)
                words.add(textFieldValue.trim())
                textFieldValue = ""
            }
        }
    }

    fun clearWords() {...}

    fun resetWords() {...}
}
```

ViewModel

kluczowe cechy i działanie viewModelScope :

- **Automatyczne Anulowanie**: To najważniejsza cecha. Każda korutyna uruchomiona w viewModelScope jest **automatycznie anulowana**, gdy ViewModel jest **niszczony** (czyli gdy użytkownik na stałe opuszcza powiązany z nim ekran).
- **Bezpieczeństwo**: Zapobiega to wyciekom pamięci i próbom **aktualizacji interfejsu**, który już **nie istnieje**. Nie ma konieczności ręcznie zarządzać zatrzymywaniem zadań w tle.
- **Domyślny Wątek**: Domyślnie viewModelScope używa Dispatchers.Main.immediate, co oznacza, że korutyna rozpoczyna pracę na wątku głównym. Jest to wygodne do szybkiej aktualizacji UI, po czym można bezpiecznie wywołać funkcję suspend, która przeniesie ciężką pracę na inny wątek (np. Dispatchers.IO).

```
class WordViewModel : ViewModel() {

    private val initialWords = listOf("Kotlin", "Compose",
    val words = mutableStateListOf<String>()
    var textFieldValue by mutableStateOf( value = "")
        private set

    init {
        resetWords()
    }
    fun onTextFieldValueChanged(newValue: String) {...}

    fun addWord() {
        if (textFieldValue.isNotBlank()) {
            viewModelScope.launch {
                delay( timeMillis = 300)
                words.add(textFieldValue.trim())
                textFieldValue = ""
            }
        }
    }

    fun clearWords() {...}

    fun resetWords() {...}
}
```

Uniwersytet Wrocławski

ViewModel

```
1 Usage
@OptIn( ...markerClass = ExperimentalMaterial3Api::class)
@Composable
fun WordListScreen(viewModel: WordViewModel = viewModel()) {
    Scaffold(
        topBar = { TopAppBar(title = { Text( text = "MVVM Word List") }) }
    ) { padding ->
        Column(...) {
            Row(...) {
                OutlinedTextField(
                    value = viewModel.textFieldValue,
                    onValueChange = viewModel::onTextFieldValueChanged,
                    label = { Text( text = "Nowe słowo") },
                    modifier = Modifier.weight( weight = 1f)
                )
                Spacer(modifier = Modifier.width( width = 8.dp))
                Button(onClick = viewModel::addWord) {
                    Text( text = "Dodaj")
                }
            }
            Spacer(modifier = Modifier.height( height = 16.dp))

            LazyColumn(modifier = Modifier.weight( weight = 1f)) {...}

            Row(...) {
                Button(onClick = viewModel::clearWords) {...}
                Button(onClick = viewModel::resetWords) {...}
            }
        }
    }
}
```

ViewModel

WordListScreen do swojego działania potrzebuje **instancji** WordViewModel, aby móc odczytywać z niej dane (stan) i wywoływać jej funkcje (akcje). Konstrukcja =viewModel() sprawia, że nie jest konieczne **ręczne tworzenie** i przekazywanie ViewModel przy **każdym wywołaniu** WordListScreen.

```
1 Usage
@OptIn( ...markerClass = ExperimentalMaterial3Api::class)
@Composable
fun WordListScreen(viewModel: WordViewModel = viewModel()) {
    Scaffold(
        topBar = { TopAppBar(title = { Text( text = "MVVM Word List") }) }
    ) { padding ->
        Column(...) {
            Row(...) {
                OutlinedTextField(
                    value = viewModel.textFieldValue,
                    onValueChange = viewModel::onTextFieldValueChanged,
                    label = { Text( text = "Nowe słowo") },
                    modifier = Modifier.weight( weight = 1f)
                )
                Spacer(modifier = Modifier.width( width = 8.dp))
                Button(onClick = viewModel::addWord) {
                    Text( text = "Dodaj")
                }
            }
            Spacer(modifier = Modifier.height( height = 16.dp))

            LazyColumn(modifier = Modifier.weight( weight = 1f)) {...}

            Row(...) {
                Button(onClick = viewModel::clearWords) {...}
                Button(onClick = viewModel::resetWords) {...}
            }
        }
    }
}
```

ViewModel

WordListScreen do swojego działania potrzebuje **instancji** WordViewModel, aby móc odczytywać z niej dane (stan) i wywoływać jej funkcje (akcje). Konstrukcja =viewModel() sprawia, że nie jest konieczne **ręczne tworzenie** i przekazywanie ViewModel przy **każdym wywołaniu** WordListScreen.

```
1 Usage
@OptIn( ...markerClass = ExperimentalMaterial3Api::class)
@Composable
fun WordListScreen(viewModel: WordViewModel = viewModel()) {
    Scaffold(
        topBar = { TopAppBar(title = { Text( text = "MVVM Word List") }) }
    ) { padding ->
        Column(...) {
            Row(...) {
                OutlinedTextField(
                    value = viewModel.textFieldValue,
                    onValueChange = viewModel::onTextFieldValueChanged,
                    label = { Text( text = "Nowe słowo") },
                    modifier = Modifier.weight( weight = 1f)
                )
                Spacer(modifier = Modifier.width( width = 8.dp))
                Button(onClick = viewModel::addWord) {
                    Text( text = "Dodaj")
                }
            }
            Spacer(modifier = Modifier.height( height = 16.dp))

            LazyColumn(modifier = Modifier.weight( weight = 1f)) {...}

            Row(...) {
                Button(onClick = viewModel::clearWords) {...}
                Button(onClick = viewModel::resetWords) {...}
            }
        }
    }
}
```

Funkcja viewModel() to specjalna funkcja z biblioteki AndroidX, która:

- Sprawdza, czy **instancja** WordViewModel **już istnieje** dla **bieżącego cyklu życia** (np. dla tego ekranu).
- Jeśli **tak**, **zwraca** tę istniejącą **instancję**.
- Jeśli **nie**, **tworzy nową instancję** i **wiąże** ją z bieżącym **cyklem życia**.

Dzięki tej funkcji, gdy użytkownik obróci ekran, WordListScreen otrzyma **tę samą** instancję WordViewModel, co wcześniej. Cały stan (lista słów) zostanie **zachowany.**

ViewModel

WordListScreen do swojego działania potrzebuje **instancji** WordViewModel, aby móc odczytywać z niej dane (stan) i wywoływać jej funkcje (akcje). Konstrukcja =viewModel() sprawia, że nie jest konieczne **ręczne tworzenie** i przekazywanie ViewModel przy **każdym wywołaniu** WordListScreen.

```
1 Usage
@OptIn( ...markerClass = ExperimentalMaterial3Api::class)
@Composable
fun WordListScreen(viewModel: WordViewModel = viewModel()) {
    Scaffold(
        topBar = { TopAppBar(title = { Text( text = "MVVM Word List") }) }
    ) { padding ->
        Column(...) {
            Row(...) {
                OutlinedTextField(
                    value = viewModel.textFieldValue,
                    onValueChange = viewModel::onTextFieldValueChanged,
                    label = { Text( text = "Nowe słowo") },
                    modifier = Modifier.weight( weight = 1f)
                )
                Spacer(modifier = Modifier.width( width = 8.dp))
                Button(onClick = viewModel::addWord) {
                    Text( text = "Dodaj")
                }
            }
            Spacer(modifier = Modifier.height( height = 16.dp))

            LazyColumn(modifier = Modifier.weight( weight = 1f)) {...}

            Row(...) {
                Button(onClick = viewModel::clearWords) {...}
                Button(onClick = viewModel::resetWords) {...}
            }
```

Funkcja viewModel() to specjalna funkcja z biblioteki AndroidX, która:

- Sprawdza, czy **instancja** WordViewModel **już istnieje** dla **bieżącego cyklu życia** (np. dla tego ekranu).
- Jeśli **tak**, **zwraca** tę istniejącą **instancję**.
- Jeśli **nie**, **tworzy nową instancję** i **wiąże** ją z bieżącym **cyklem życia**.

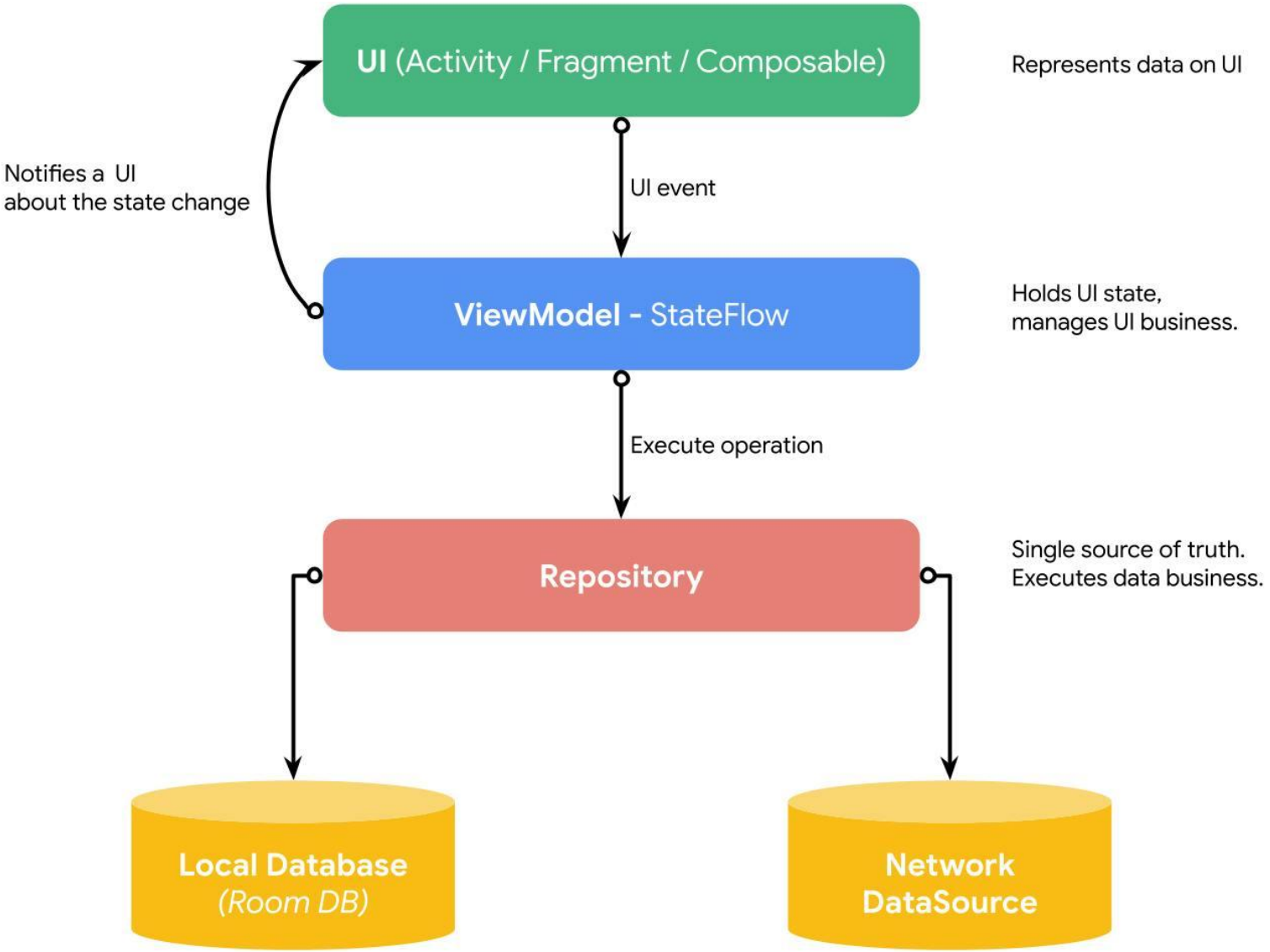
Dzięki tej funkcji, gdy użytkownik obróci ekran, WordListScreen otrzyma **tę samą** instancję WordViewModel, co wcześniej. Cały stan (lista słów) zostanie **zachowany.**

Znak **::** tworzy **referencję do funkcji** (lub właściwości). Jest to zwięzły sposób na **przekazanie funkcji** jako **parametru.**

onValueChange = { nowyTekst -> viewModel.onTextFieldValueChanged(nowyTekst) }

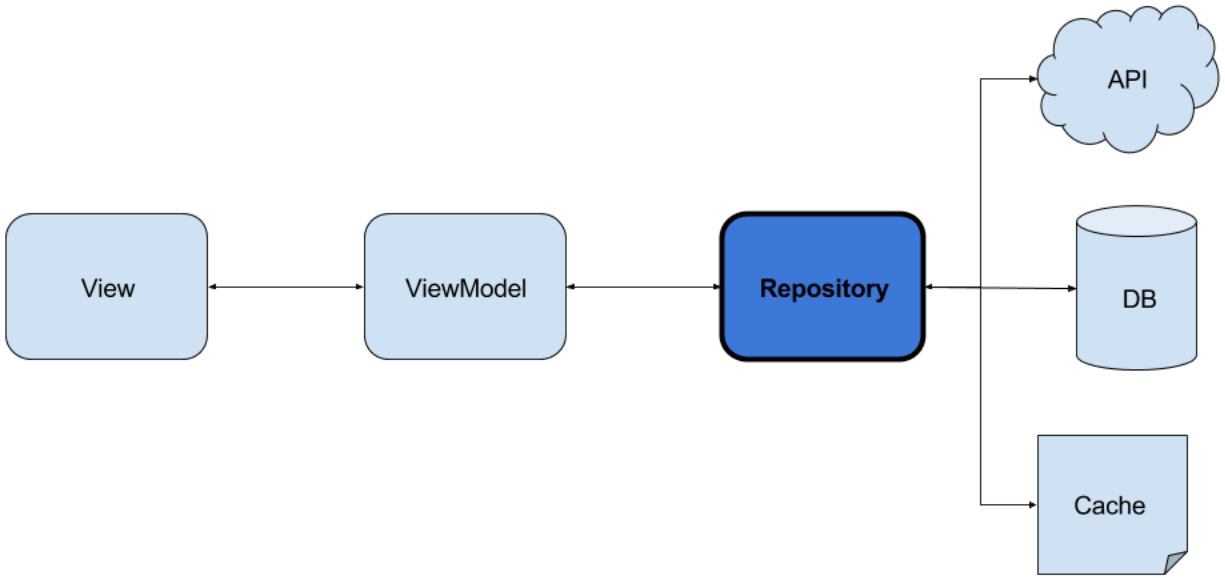
Uniwersytet Wrocławski

Repozytorium



Repozytorium

W naszej aplikacji ViewModel sam **przechowywał listę słów** i **logikę jej modyfikacji**. Na początku to działa, ale co, jeśli te dane miałyby pochodzić z pliku, bazy danych lub internetu? ViewModel musiałby „*wiedzieć*”, jak rozmawiać z każdym z tych źródeł. Staje się zbyt skomplikowany i łamie zasadę jednej odpowiedzialności.



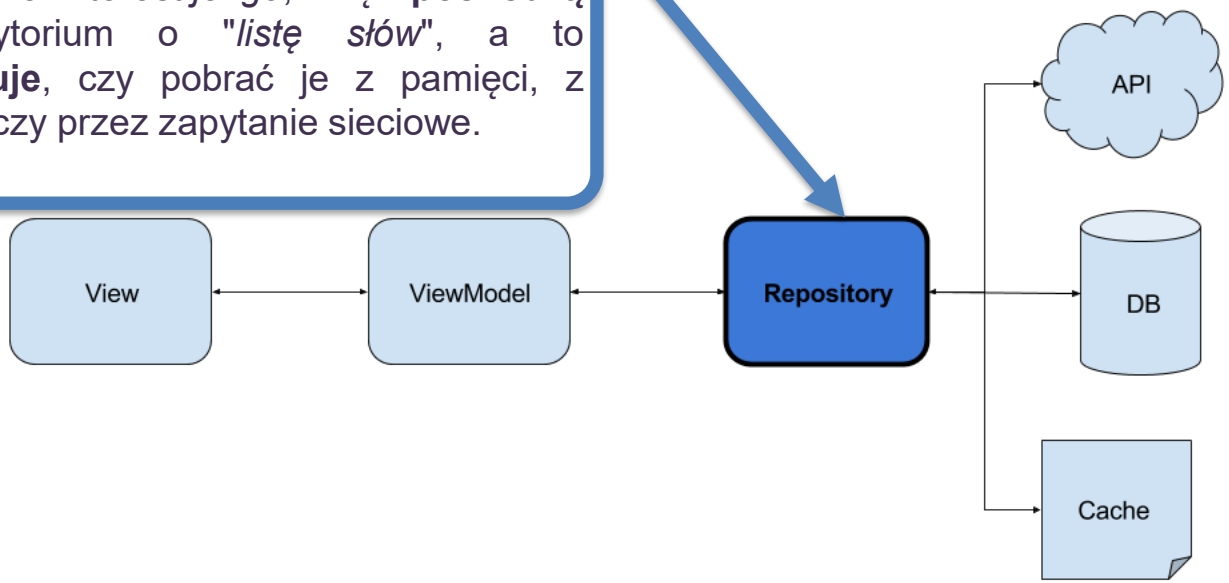
Repozytorium

W naszej aplikacji ViewModel sam **przechowywał listę słów** i **logikę jej modyfikacji**. Na początku to działa, ale co, jeśli te dane miałyby pochodzić z pliku, bazy danych lub internetu? ViewModel musiałby „*wiedzieć*”, jak rozmawiać z każdym z tych źródeł. Staje się zbyt skomplikowany i łamie zasadę jednej odpowiedzialności.

Nasz Szef Kuchni (ViewModel) jest świetny w gotowaniu (zarządzaniu logiką UI), ale obecnie sam chodzi na targ (API), do spiżarni (baza danych) i do hurtowni (cache).

Tutaj wprowadzamy specjalistę - **Repozytorium**. Jest to klasa, która działa jak **pośrednik** między **logiką aplikacji** (ViewModel) a różnymi **źródłami danych.**

Repozytorium jest **jedynym miejscem** w aplikacji, z którym ViewModel *rozmawia*, gdy **potrzebuje** danych. ViewModel *nie wie* i *nie interesuje go*, skąd **pochodzą dane**. Pyta Repozytorium o "*listę słów*", a to Repozytorium **decyduje**, czy pobrać je z pamięci, z lokalnej bazy danych, czy przez zapytanie sieciowe.



Repozytorium

Dodajmy Repozytorium do projektu.

```
class WordRepository {
    // Dane i logika zostały przeniesione tutaj
    1 Usage
    private val initialWords = listOf("Kotlin", "Compose", "Architektura", "Repozytorium")

    1 Usage
    suspend fun getInitialWords(): List<String> {
        delay( timeMillis = 500) // Symulacja opóźnienia dostępu do danych
        return initialWords
    }

    1 Usage
    suspend fun addWord(currentList: List<String>, word: String): List<String> {
        delay( timeMillis = 300)
        return currentList + word
    }
}
```

Repozytorium

Musimy wprowadzić zmiany w ViewModel

ViewModel teraz **wymaga dostarczenia** mu instancji WordRepository do swojego działania. Stał się **zależny** od warstwy danych, co jest fundamentem **wstrzykiwania zależności**.

ViewModel *nie interesuje* w jaki **sposób** Repozytorium zdobędzie dane – wysyła tylko **żądanie**.

```
class WordViewModel(private val wordRepository: WordRepository) : ViewModel() {
    7 Usages
    val words = mutableStateListOf<String>()
    5 Usages
    var textFieldValue by mutableStateOf( value = ""); private set

    init {
        resetWords()
    }
    1 Usage
    fun onTextFieldValueChanged(newValue: String) {...}
    1 Usage
    fun addWord() {
        if (textFieldValue.isNotBlank()) {
            viewModelScope.launch {
                val updatedList = wordRepository.addWord(
                    currentList = words,
                    word = textFieldValue.trim())
                words.clear()
                words.addAll( elements = updatedList)
                textFieldValue = ""
            }
        }
    }
    2 Usages
    fun resetWords() {...}
    1 Usage
    fun clearWords() {...}
}
```

Uniwersytet Wrocławski

Repozytorium

Teraz ViewModel wymaga dostarczenia **instancji repozytorium** do konstruktora – domyślnie ViewModel posiada **pusty konstrutor** – konieczne jest stworzenie własnej **fabryki**.

Konieczne jest dodanie do projektu, ponieważ **domyślny mechanizm** tworzenia ViewModel w Androidzie potrafi tworzyć tylko te, które mają **pusty konstruktor**.

```
class WordViewModelFactory(private val repository: WordRepository) : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom( cls = WordViewModel::class.java)) {
            @Suppress( ...names = "UNCHECKED_CAST")
            return WordViewModel( wordRepository = repository) as T
        }
        throw IllegalArgumentException( s = "Unknown ViewModel class")
    }
}
```

Uniwersytet Wrocławski

Repozytorium

Teraz ViewModel wymaga dostarczenia **instancji repozytorium** do konstruktora – domyślnie ViewModel posiada **pusty konstrutor** – konieczne jest stworzenie własnej **fabryki**.

Konieczne jest dodanie do projektu, ponieważ **domyślny mechanizm** tworzenia ViewModel w Androidzie potrafi tworzyć tylko te, które mają **pusty konstruktor**.

```
class WordViewModel(private val wordRepository: WordRepository) : ViewModel() {
```

Gdy dodaliśmy WordRepository jako **parametr do konstruktora** WordViewModel, konieczne jest dostarczenie ***instrukcji*** do tworzenia WordViewModel.

```
class WordViewModelFactory(private val repository: WordRepository) : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        if (modelClass.isAssignableFrom( cls = WordViewModel::class.java)) {
            @Suppress( ...names = "UNCHECKED_CAST")
            return WordViewModel( wordRepository = repository) as T
        }
        throw IllegalArgumentException( s = "Unknown ViewModel class")
    }
}
```

Repozytorium

Tworzymy nową instancję WordRepository

Tworzymy **fabrykę** ViewModelu i Przekazujemy do niej **repozytorium.**

Wywołujemy funkcję viewModel(), ale tym razem **przekazujemy do niej** naszą własną **fabrykę**. Zostanie ona wykorzystana do **stworzenia** (lub pobrania istniejącej) **instancję** WordViewModel, poprawnie **dostarczając** mu WordRepository.

```
@OptIn( ...markerClass = ExperimentalMaterial3Api::class)
@Composable
fun WordListScreen() {
    val repository = WordRepository()
    val factory = WordViewModelFactory(repository)
    val viewModel: WordViewModel = viewModel(factory = factory)
```