Uniwersytet Wrocławski

Rafał Lewandków

PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

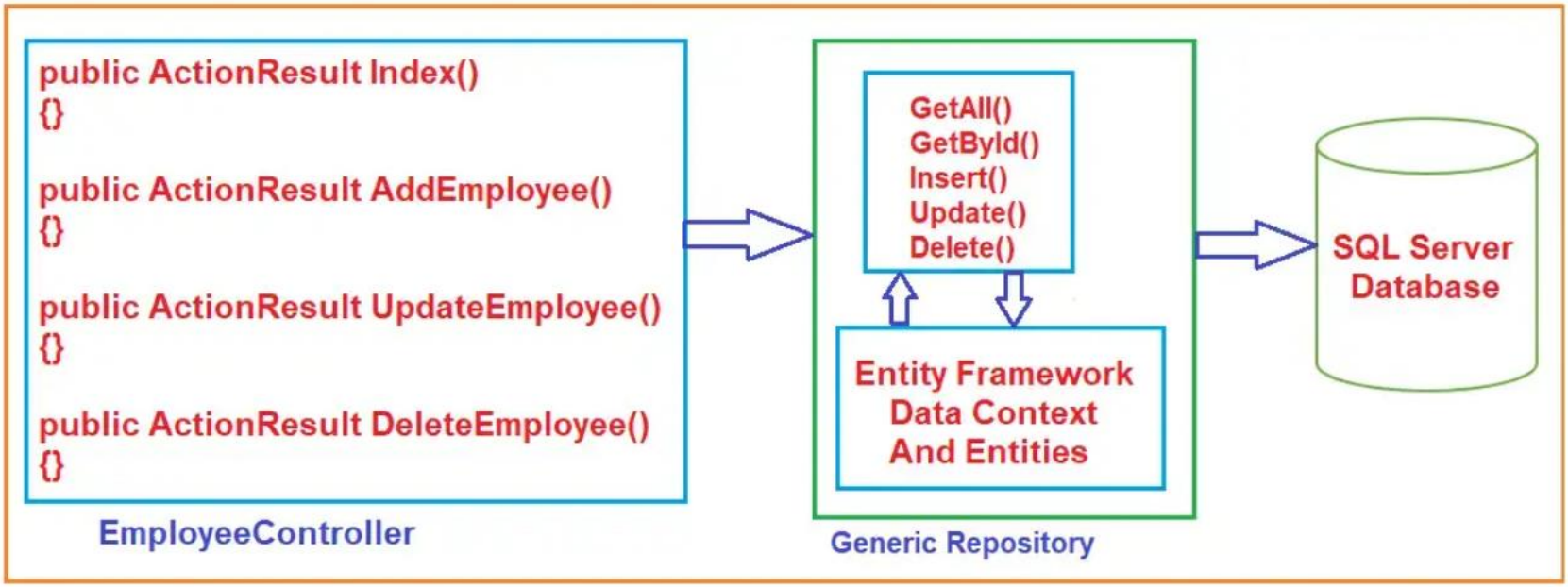
WYKŁAD 10

Lokalne przechowywanie danych

- Wzorzec Repozytorium
- SharedPreferences
- DataStore

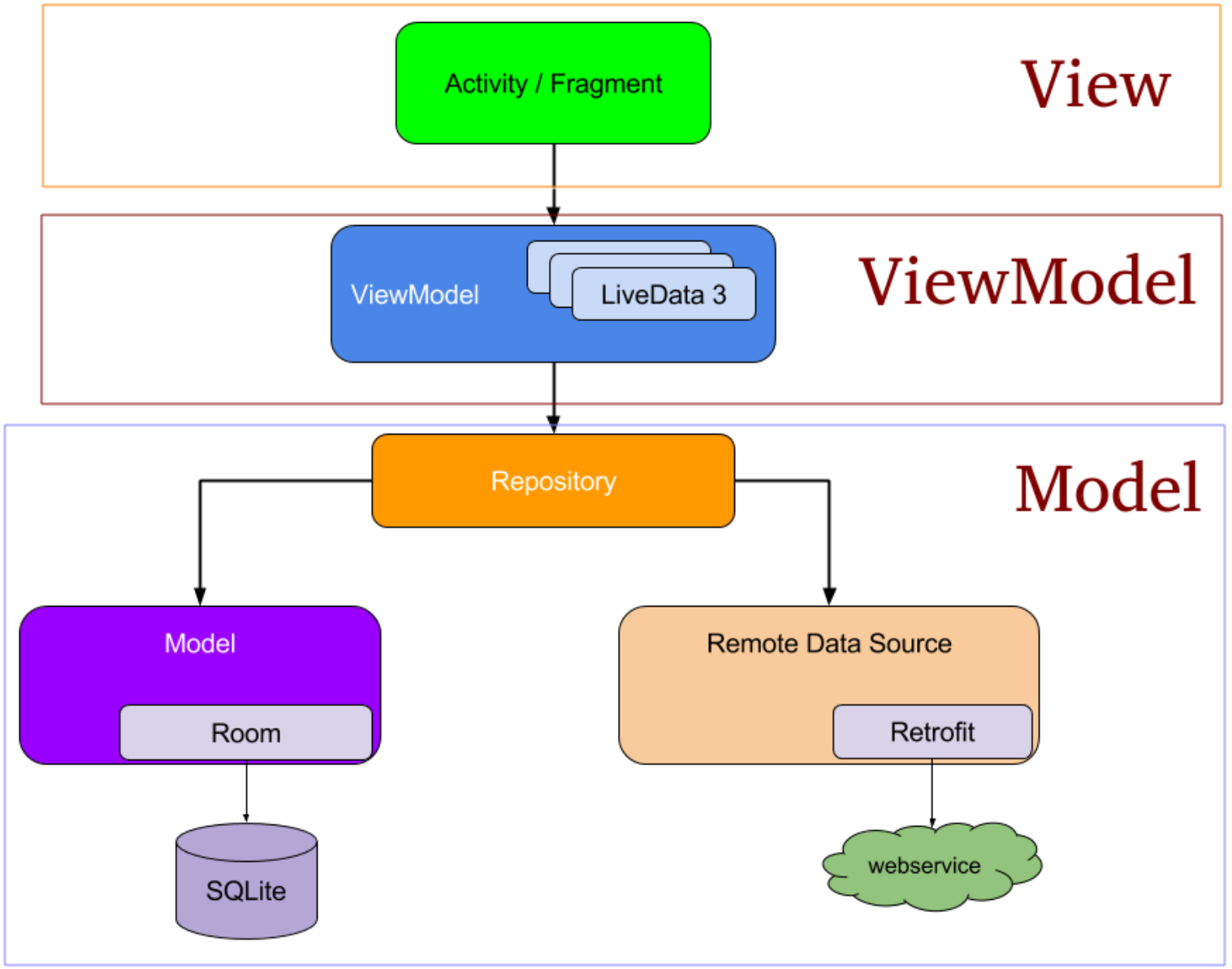
Uniwersytet Wrocławski

Wzorzec Repozytorium



https://dotnettutorials.net/wp-content/uploads/2018/07/Why-do-we-need-Generic-Repository-Pattern-in-C.png

Wzorzec Repozytorium



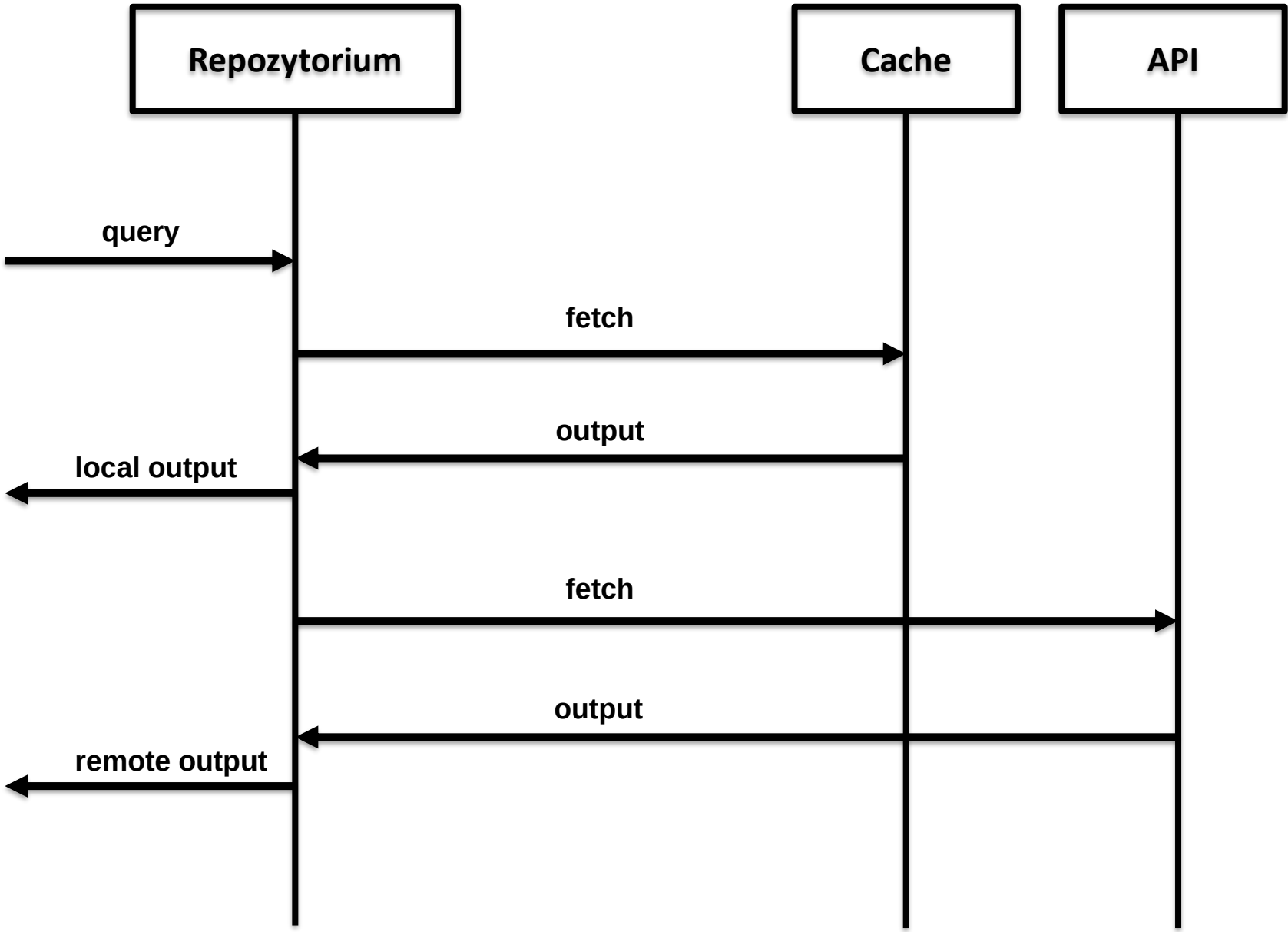
Wzorzec Repozytorium

Repozytorium jest warstwą pośredniczącą między danymi a resztą aplikacji. W kontekście MVVM repozytorium pomaga w oddzieleniu logiki biznesowej od interakcji z danymi:

- **Separacja odpowiedzialności:** Izoluje logikę dostępu do danych
- **Łatwa zmiana źródeł danych:** Jeżeli nasza aplikacja korzysta z lokalnej bazy danych, ale chcemy przejść na zdalne API, możemy to zrobić bez konieczności modyfikowania ViewModel
- **Optymalizacja dostępu do danych:** Można wprowadzić dodatkowe mechanizmy dostępu do danych np. *offline caching*
- **Logika dostępu do danych:** w przypadku kilku źródeł danych

Uniwersytet Wrocławski

Wzorzec Repozytorium



Wzorzec Repozytorium

```
class UserRepository {
    suspend fun getUsers(): List<User> {
        delay(700L)
        return DataProvider.users
    }
}
```

```
class UserViewModel : ViewModel() {

    private val userRepository = UserRepository()

    private val _usersList = MutableStateFlow<List<User>>(emptyList())
    val usersList: StateFlow<List<User>> get() = _usersList

    init {
        loadUsers()
    }

    private fun loadUsers() {
        viewModelScope.launch {
            _usersList.value = userRepository.getUsers()
        }
    }
}
```

SharedPreferences vs DataStore

- **Typ daych i bezpieczeństwo:**
  - **SharedPreferences** przechowuje dane w plikach **XML** i obsługuje tylko typy proste
  - **DataStore** obsługuje również niestandardowe typy danych i zapewnia automatyczną obsługę konwersji do i z formatu **protobuf**

- **Wsparcie dla asynchroniczności:**
  - **SharedPreferences** oferuje tylko operacje **synchroniczne**
  - **DataStore** został zaprojektowany z myślą o operacjach **asynchronicznych** - wspiera Coroutines

- **Bezpieczeństwo wątkowe:**
  - **SharedPreferences** nie posiada żadnych wbudowanych mechanizmów
  - **DataStore** posiada wbudowane mechanizmy bezpieczeństwa ze względu na wielowątkowość

- **Obsługa zmian danych:**
  - **SharePreferences** nie oferuje wbudowanej obsługi reagowania na zmiany danych
  - **DataStore** umożliwia korzystanie z **LiveData** i **Flow** oferując pełne wsparcie

Uniwersytet Wrocławski

SharedPreferences

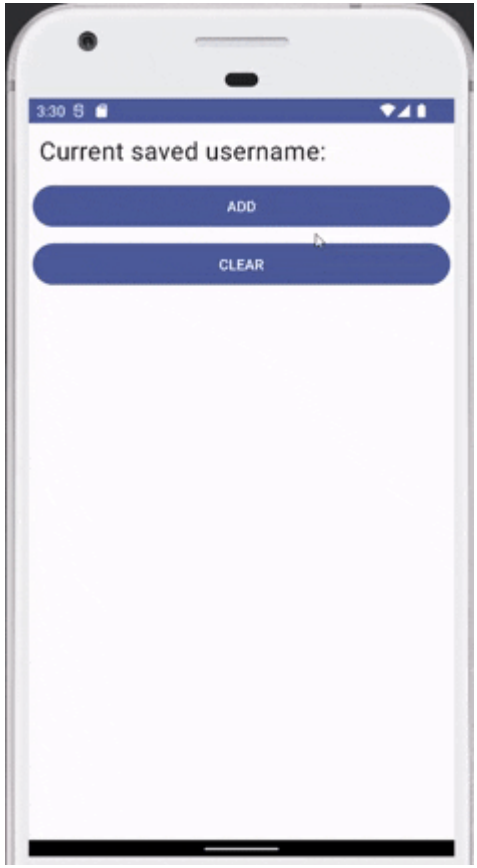
```
class UserRepository(application: Application) {

    private val sharedPref = application.getSharedPreferences("fileName", MODE_PRIVATE)

    private var _username: String = sharedPref.getString("username", "") ?: ""
    val username: String
        get() = _username

    fun add(newUsername: String) {
        val edit = sharedPref.edit()
        edit.putString("username", newUsername).apply()
        _username = newUsername
    }

    fun clear() {
        val edit = sharedPref.edit()
        edit.putString("username", "").apply()
        _username = ""
    }
}
```



`private val sharedPref = application.getSharedPreferences("fileName", MODE_PRIVATE)` - Wykorzystuje `getSharedPreferences` na obiekcie `Application`, aby uzyskać dostęp do `SharedPreferences` o nazwie `fileName`. Argument `MODE_PRIVATE` wskazuje, że `SharedPreferences` są prywatne dla tej aplikacji i nie są dostępne dla innych aplikacji. Inne tryby:

- `MODE_APPEND` - pozwala dopisywać kolejne elementy bez nadpisywania
- `MODE_PRIVATE` - najczęściej wykorzystywany, dostęp do pliku tylko z poziomu aplikacji
- `MODE_WORLD_READABLE` - zezwala innym aplikacjom na odczyt
- `MODE_WORLD_WRITABVLE` - zezwala innym aplikacjom na zapis

ViewModel + XML

```
class UserViewModel(application: Application) : AndroidViewModel(application) {
    private val repository: UserRepository = UserRepository(application)
    private val _username: MutableStateFlow<String> = MutableStateFlow(repository.username)

    val username: StateFlow<String>
        get() = _username

    fun addUsername(username: String) {
        repository.add(username)
        _username.value = username
    }

    fun clearUsername() {
        repository.clear()
        _username.value = ""
    }
}
```

Uniwersytet Wrocławski

ViewModel + Compose

```
class UserViewModelFactory(private val application: Application) :
    ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return UserViewModel(application) as T
    }
}
```

```
class UserViewModel(application: Application) : ViewModel() {
    private val repository: UserRepository = UserRepository(application)
    private val _username: MutableStateFlow<String> = MutableStateFlow(repository.username)

    val username: StateFlow<String>
        get() = _username

    fun addUsername(username: String) {
        repository.add(username)
        _username.value = username
    }

    fun clearUsername() {
        repository.clear()
        _username.value = ""
    }
}
```

```
val viewModel: UserViewModel = viewModel(
    LocalViewModelStoreOwner.current!!,
    "UserViewModel",
    UserViewModelFactory(LocalContext.current.applicationContext as Application)
)
```

Uniwersytet Wrocławski

DataStore

Oferuje dwa główne rodzaje implementacji:

- `Preferences DataStore` - Ten rodzaj jest podobny do `SharedPreferences`, ale jest oparty na protokole `Kotlin Coroutines` i zapewnia bezpieczne przechowywanie danych. Można w nim przechowywać dane w postaci *klucz-wartość*, gdzie klucze są ciągami znaków, a wartości mogą być różnymi typami danych, takimi jak liczby, ciągi znaków itp.
- `Proto DataStore` - Ten rodzaj pozwala na zapisywanie danych w formacie `protobuf`. Jest to format serializacji danych opracowany przez firmę Google.

```
object SaveUsernameDataStore {
    private val Context.dataStore: DataStore<Preferences> by preferencesDataStore("user_prefs")
    private val USERNAME_KEY = stringPreferencesKey("USERNAME")

    suspend fun storeUsername(context: Context, username: String) {
        context.dataStore.edit { preferences ->
            preferences[USERNAME_KEY] = username
        }
    }

    fun getUsernameFlow(context: Context): Flow<String> {
        return context.dataStore.data.map { preferences ->
            preferences[USERNAME_KEY] ?: ""
        }
    }
}
```

`private val Context.dataStore: DataStore<Preferences> by preferencesDataStore("user_prefs")` - Ta linia definiuje rozszerzenie (extension property) dla klasy `Context`. Tworzy ono obiekt `DataStore` typu `Preferences` o nazwie `user_prefs`. Ten `DataStore` będzie używany do przechowywania i pobierania preferencji użytkownika.

`private val USERNAME_KEY = stringPreferencesKey("USERNAME")` - Ta linia definiuje prywatną stałą, która reprezentuje klucz używany do przechowywania i pobierania nazwy użytkownika w `DataStore`. Jest to obiekt typu `Preferences.Key<String>`.

Uniwersytet Wrocławski

DataStore

```
class UserRepository(private val application: Application) {
    fun getUsername() = SaveUsernameDataStore.getUsernameFlow(application)
    suspend fun add(username: String) = SaveUsernameDataStore.storeUsername(application, username)
    suspend fun clear() = SaveUsernameDataStore.storeUsername(application, "")
}
```

```
class UserViewModel(application: Application) : AndroidViewModel(application) {
    private val repository: UserRepository
    private val _username = MutableStateFlow("")
    val username: StateFlow<String>
        get() = _username

    init {
        repository = UserRepository(application)
        fetchUser()
    }

    private fun fetchUser() {
        viewModelScope.launch {
            repository.getUsername().collect { username ->
                _username.value = username
            }
        }
    }

    fun addUsername(username: String) {
        viewModelScope.launch {
            repository.add(username)
        }
    }

    fun clearUsername(){
        viewModelScope.launch {
            repository.clear()
        }
    }
}
```