Uniwersytet Wrocławski

Rafał Lewandków

# PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

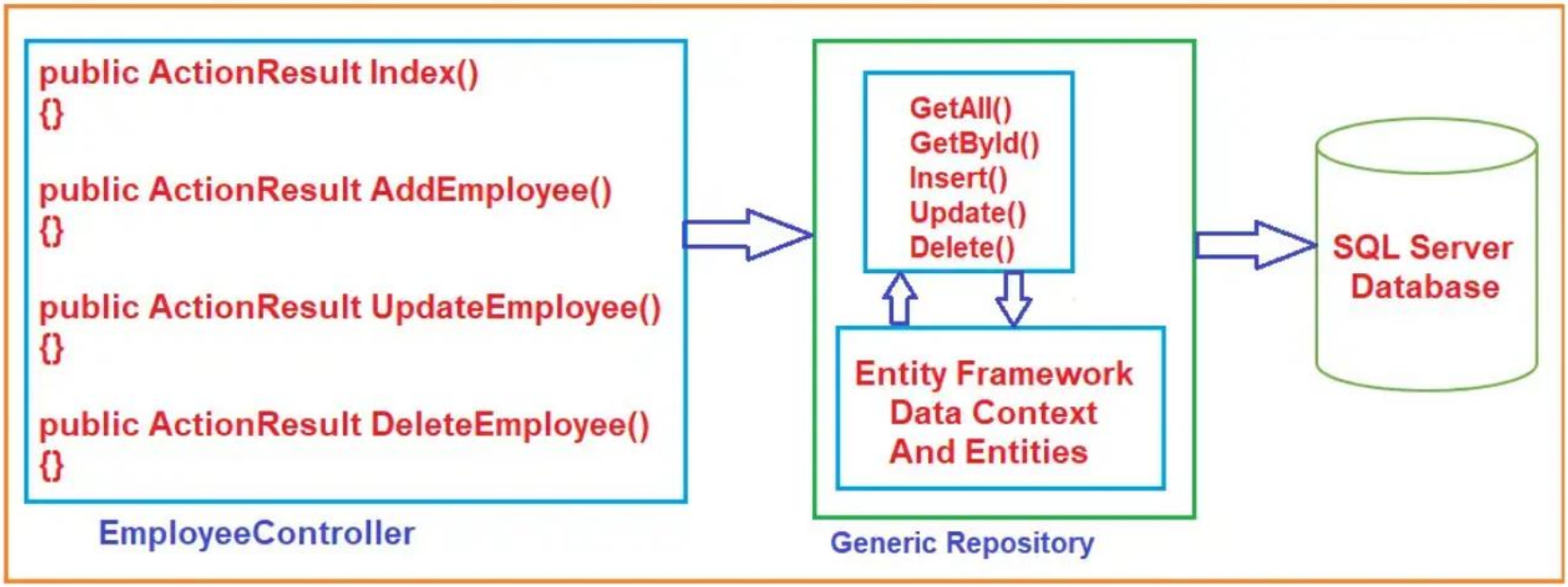
## WYKŁAD 10

Lokalne przechowywanie danych

- Wzorzec Repozytorium
- SharedPreferences
- DataStore

Uniwersytet Wrocławski

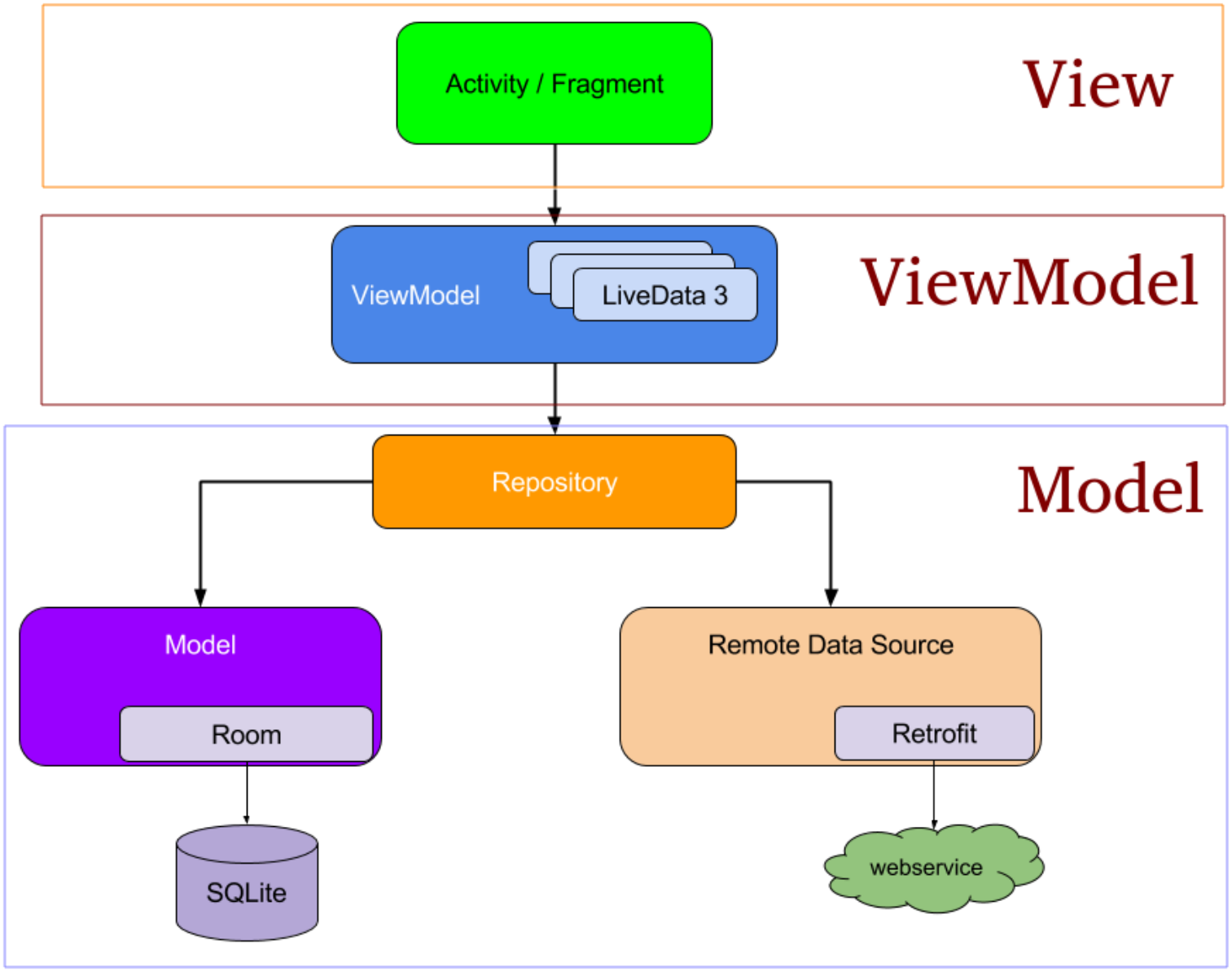
# Wzorzec Repozytorium




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

Uniwersytet Wrocławski

# Wzorzec Repozytorium



Uniwersytet Wrocławski

# Wzorzec Repozytorium

**Repozytorium** jest warstwą pośredniczącą między danymi a resztą aplikacji, zapewniającymi dostęp do źródeł danych (np. bazy danych, zdalne API, odczyt z pliku) oraz operacje na danych. W kontekście wzorca `MVVM`, repozytorium pomaga w oddzieleniu logiki biznesowej od interakcji z danymi, co ułatwia testowanie, utrzymanie i rozszerzanie aplikacji.

- Separacja odpowiedzialności - Izoluje logikę dostępu do danych od logiki biznesowej i ui. To pozwala na łatwiejsze zarządzanie i utrzymanie kodu, a także ułatwia współpracę między różnymi zespołami programistycznymi.
- Testowalność - Zapewnia łatwość testowania, ponieważ zapewnia abstrakcję nad danymi. Dzięki temu możemy tworzyć testy jednostkowe i testy integracyjne, które pozwalają nam weryfikować poprawność funkcjonalności bez konieczności dostępu do rzeczywistych źródeł danych.
- Łatwa wymiana źródeł danych - Dzięki repozytorium możemy zmieniać źródło danych bez wprowadzania zmian w innych częściach aplikacji. Na przykład, jeśli nasza aplikacja korzysta z lokalnej bazy danych, ale chcemy w przyszłości przejść na zdalne API, możemy to zrobić bez konieczności modyfikowania `ViewModel` 'ów i warstwy ui.
- Obsługa błędów i odzyskiwanie - Repozytoria mogą obsługiwać błędy związane z danymi, takie jak problemy z siecią czy błędy związane z bazą danych.
- Optymalizacja dostępu do danych - Repozytoria mogą wprowadzać mechanizmy optymalizacji dostępu do danych, np. *cache'owanie* wyników zapytań czy ograniczenie ilości komunikacji sieciowej. To może przyspieszyć działanie aplikacji i zmniejszyć zużycie zasobów urządzenia. (*cachowanie* pojawi się w ostatnim module zajęć)
- Logika dostępu do danych - jeżeli mamy kilka źródeł danych, logikę dostępu (dostęp do API co określony czas, na żądanie użytkownika, w przeciwnym wypadku wczytanie danych z lokalnej bazy) możemy umieścić w repozytorium, pozostawiając `ViewModel` z jednym źródłem danych

Uniwersytet Wrocławski

# Wzorzec Repozytorium

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

Uniwersytet Wrocławski

# 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()
        }
    }
}
```

# RecyclerView

```
class UserViewHolder(private val binding: RvItemBinding) : RecyclerView.ViewHolder(binding.root) {
    fun bind(item: User) {
        val fullName = "${item.firstName} ${item.lastName}"
        binding.userTextView.text = fullName
    }
}
```

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

    override fun areContentsTheSame(oldItem: User, newItem: User): Boolean {
        return oldItem == newItem
    }
}
```

```
class UserAdapter(userComparator: UserComparator) : ListAdapter<User, UserViewHolder>(userComparator) {
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): UserViewHolder {
        return UserViewHolder(
            RvItemBinding.inflate(
                LayoutInflater.from(parent.context), parent, false
            )
        )
    }

    override fun onBindViewHolder(holder: UserViewHolder, position: Int) {
        val item = getItem(position)
        holder.bind(item)
    }
}
```

# Fragment

```
viewLifecycleOwner.lifecycleScope.launch {
    viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED){
        viewModel.usersList.collectLatest{ users ->
            userAdapter.submitList(users)
        }
    }
}

binding.rvList.apply{
    adapter = userAdapter
    layoutManager = LinearLayoutManager(requireContext())
}
```

# Fragment

```
viewLifecycleOwner.lifecycleScope.launch {
    viewLifecycleOwner.repeatOnLifecycle(Lifecycle.State.STARTED){
        viewModel.usersList.collectLatest{ users ->
            userAdapter.submitList(users)
        }
    }
}

binding.rvList.apply{
    adapter = userAdapter
    layoutManager = LinearLayoutManager(requireContext())
}
```

# Compose

```
@Composable
fun MainScreen() {

    val viewModel: UserViewModel = viewModel()
    val users by viewModel.usersList.collectAsStateWithLifecycle()

    LazyColumn() {
        items(users.size) {
            Text(
                text = "${users[it].firstName} ${users[it].lastName}",
                fontSize = 32.sp,
                textAlign = TextAlign.Center,
                modifier = Modifier
                    .fillMaxWidth()
                    .padding(2.dp)
            )
        }
    }
}
```

Uniwersytet Wrocławski

# SharedPreferences vs DataStore

- Typ danych i bezpieczeństwo:
  - `SharedPreferences` przechowuje dane w plikach `XML` i obsługuje tylko typy prostych danych, takie jak liczby całkowite, ciągi znaków, wartości boolean itp. Jednak nie oferuje wbudowanej obsługi bardziej złożonych struktur danych.
  - `DataStore` obsługuje niestandardowe typy danych i zapewnia automatyczną obsługę konwersji do i z formatu `protobuf`.
- Wsparcie dla asynchroniczności:
  - `SharedPreferences` oferuje operacje synchroniczne, co może wpływać na wydajność aplikacji, szczególnie gdy operacje odczytu/zapisu danych są wykonywane w wątku głównym.
  - `DataStore` został zaprojektowany z myślą o asynchroniczności i wspiera Coroutines, dzięki czemu operacje odczytu/zapisu danych mogą być wykonywane asynchronicznie, co poprawia wydajność i responsywność aplikacji.
- Bezpieczeństwo wątkowe:
  - `SharedPreferences` nie jest wątkowo bezpieczne, co oznacza, że operacje odczytu/zapisu mogą powodować błędy synchronizacji, jeśli są wykonywane równocześnie przez wiele wątków.
  - `DataStore` posiada wbudowane mechanizmy bezpieczeństwa ze względu na wielowątkowość. Można go bezpiecznie używać w aplikacjach wielowątkowych bez konieczności dodatkowej synchronizacji.
- Obsługa zmian danych:
  - `SharedPreferences` nie oferuje wbudowanej obsługi reagowania na zmiany danych. Możemy jedynie odczytać aktualny stan.
  - `DataStore` umożliwia korzystanie z obiektu `Flow` lub `LiveData`, co pozwala na automatyczną obsługę zmian danych. Możemy zarejestrować obserwatora, który będzie otrzymywał powiadomienia o zmianach danych, co ułatwia reagowanie na aktualizacje danych w czasie rzeczywistym.

Uniwersytet Wrocławski

# SharedPreferences

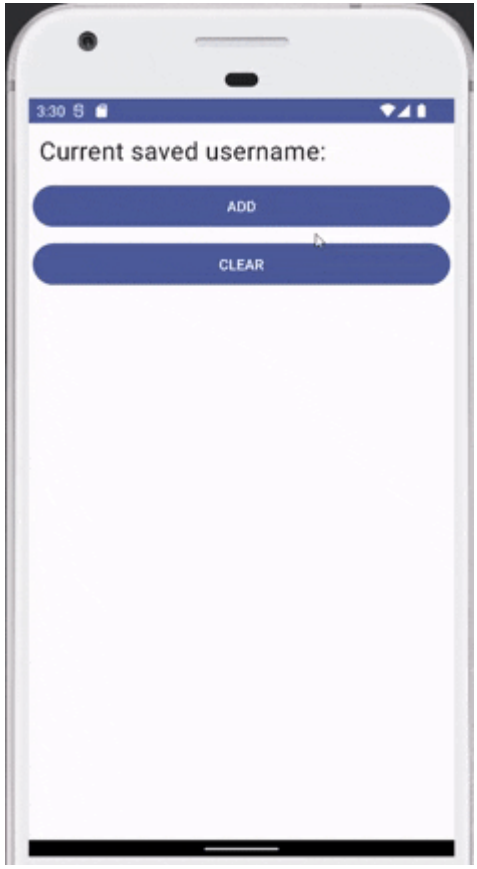
```
class UserRepository(application: Application) {

    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>`.

# 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()
        }
    }
}
```