Uniwersytet Wrocławski

Rafał Lewandków

PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 2

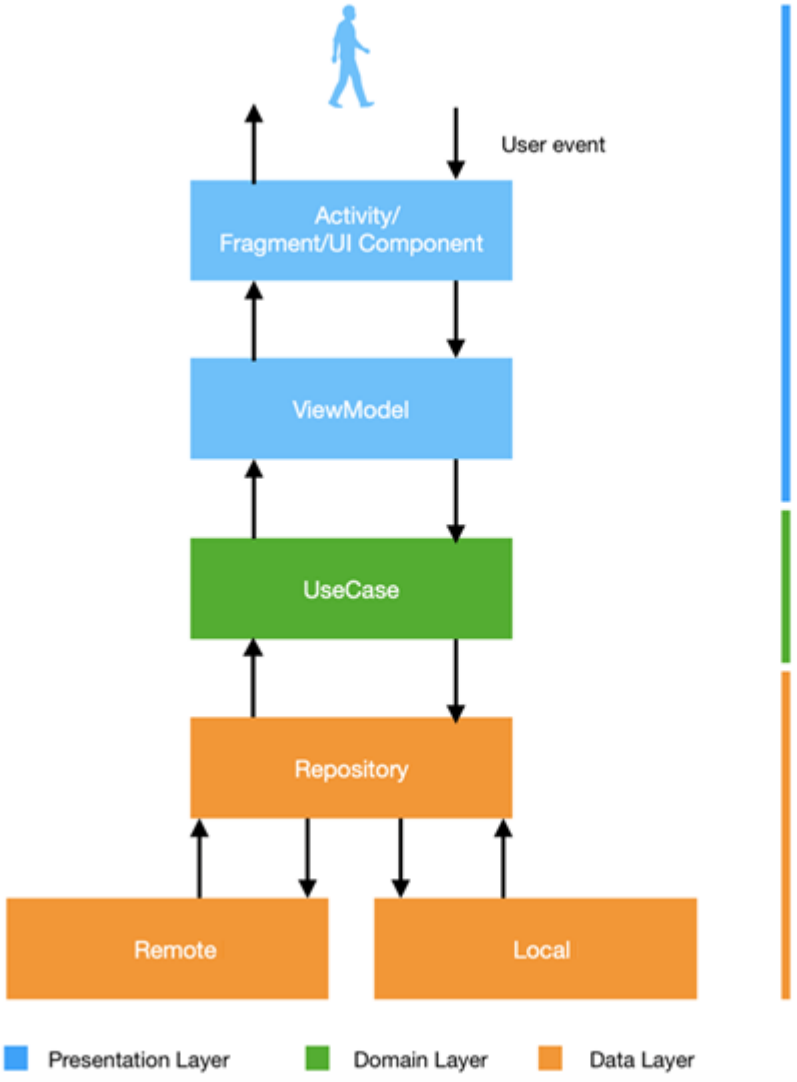
WYKŁAD 13

Wzorzec Single Source of Truth

- Strategia Offline Caching
- Mutex

Single Source of Truth

Możemy pobierać dane z sieci za pomocą Retrofit i jak zapisywać je lokalnie w bazie Room. Jedną ze strategii jest wykorzystanie obu tych elementów aby zbudować aplikację, która działa nawet **bez dostępu do internetu**, implementując wzorzec **Single Source of Truth (SSoT) ze strategią Offline Caching.**



Single Source of Truth

Możemy pobierać dane z sieci za pomocą Retrofit i jak zapisywać je lokalnie w bazie Room. Jedną ze strategii jest wykorzystanie obu tych elementów aby zbudować aplikację, która działa nawet **bez dostępu do internetu**, implementując wzorzec **Single Source of Truth (SSoT) ze strategią Offline Caching.**

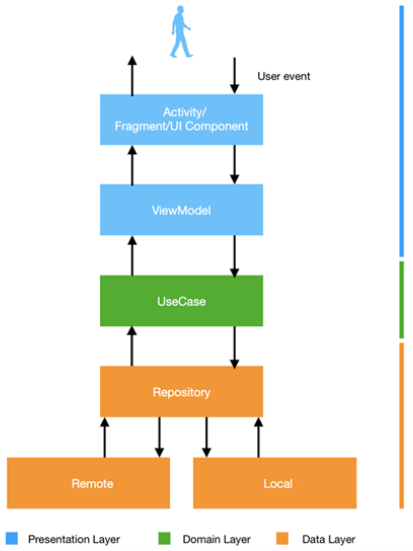
W SSoT, UI **zawsze** czyta dane tylko **z jednego, lokalnego źródła prawdy**, którym jest baza danych (Room). UI nigdy nie widzi danych prosto z sieci.

Ścieżka Zapisu/Synchronizacji (dla Danych z Sieci):

Retrofit (API) —(pobierz dane)→ Repository —(zapisz dane)→ Room (Baza Danych)

Ścieżka Odczytu (dla UI):

UI ←(Flow)— ViewModel ←(Flow)— Repository ←(Flow)— Room (Baza Danych)



Uniwersytet Wrocławski

Offline Caching

```
class UserRepository @Inject constructor(
    private val apiService: RandomUserApiService,
    private val userDao: UserDao
) {
    // Jedyne źródło prawdy to baza danych
    1 Usage
    val usersStream: Flow<List<UserEntity>> = userDao.getUsersStream()

    1 Usage
    suspend fun refreshUsers() {
        try {
            val response = apiService.getUsers()
            val entities = response.results.map { dto ->
                UserEntity(
                    uuid = dto.login.uuid,
                    firstName = dto.name.first,
                    lastName = dto.name.last,
                    email = dto.email
                )
            }
            // Czyścimy starą bazę i wstawiamy nowe, świeże dane
            userDao.clearAll()
            userDao.upsertUsers( users = entities)
        } catch (e: Exception) {
            Log.e( tag = "UserRepository",  msg = "Failed to fetch users: ${e.message}")
            throw e
        }
    }
}
```

Uniwersytet Wrocławski

Offline Caching

Jak połączyć **dane z serwera** ze **stanem**, który jest **modyfikowany lokalnie** przez użytkownika (np. Dodanie elementu do *ulubionych*)? Głównym wyzwaniem jest zachowanie stanu *ulubionych* podczas odświeżania danych z API, które o tym stanie nic nie wie.

Najprostsze rozwiązanie: Repository najpierw sprawdzi, którzy użytkownicy są obecnie oznaczeni jako ulubieni, a następnie zachowa ten stan, gdy wstawi nowe dane z sieci**.**

Uniwersytet Wrocławski

Offline Caching

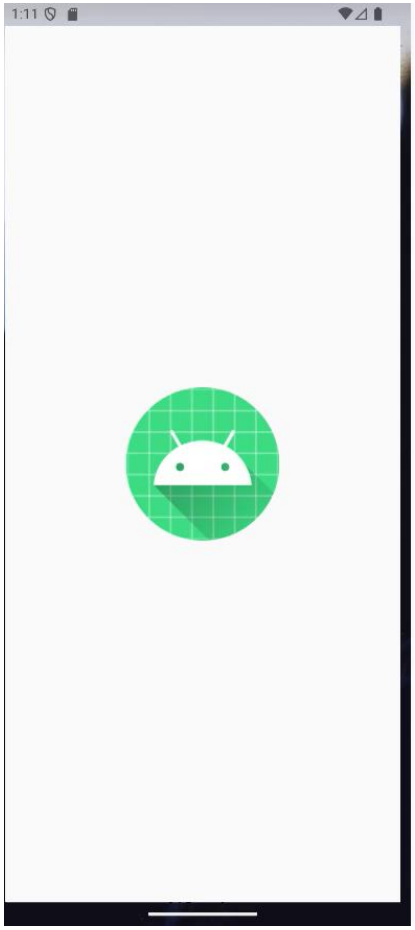
```
class UserRepository @Inject constructor(
    private val userDao: UserDao
) {
    val usersStream: Flow<List<UserEntity>> = userDao.getUsersStream()

    1 Usage
    suspend fun toggleFavoriteStatus(uuid: String) {
        val user = userDao.getUserByUuid(uuid)
        user?.let {
            userDao.updateUser( user = it.copy(isFavorite = !it.isFavorite))
        }
    }

    suspend fun refreshUsers() {
        try {
            // 1. Pobierz z bazy listę ID ulubionych użytkowników ZANIM ją wyczyścisz
            val favoriteUserIds = userDao.getUsersStream().first()
                .filter { it.isFavorite }
                .map { it.uuid }
                .toSet()

            // 2. Pobierz nowe dane z sieci
            val response = apiService.getUsers()
            val entities = response.results.map { dto ->
                UserEntity(
                    uuid = dto.login.uuid,
                    firstName = dto.name.first,
                    lastName = dto.name.last,
                    email = dto.email,
                    // 3. Sprawdź, czy nowy użytkownik był na liście ulubionych
                    isFavorite = favoriteUserIds.contains(dto.login.uuid)
                )
            }

            // 4. Wyczyść starą bazę i wstaw nowe dane z zachowanym stanem ulubionych
            userDao.clearAll()
            userDao.upsertUsers( users = entities)
```

Uniwersytet Wrocławski

Offline Caching

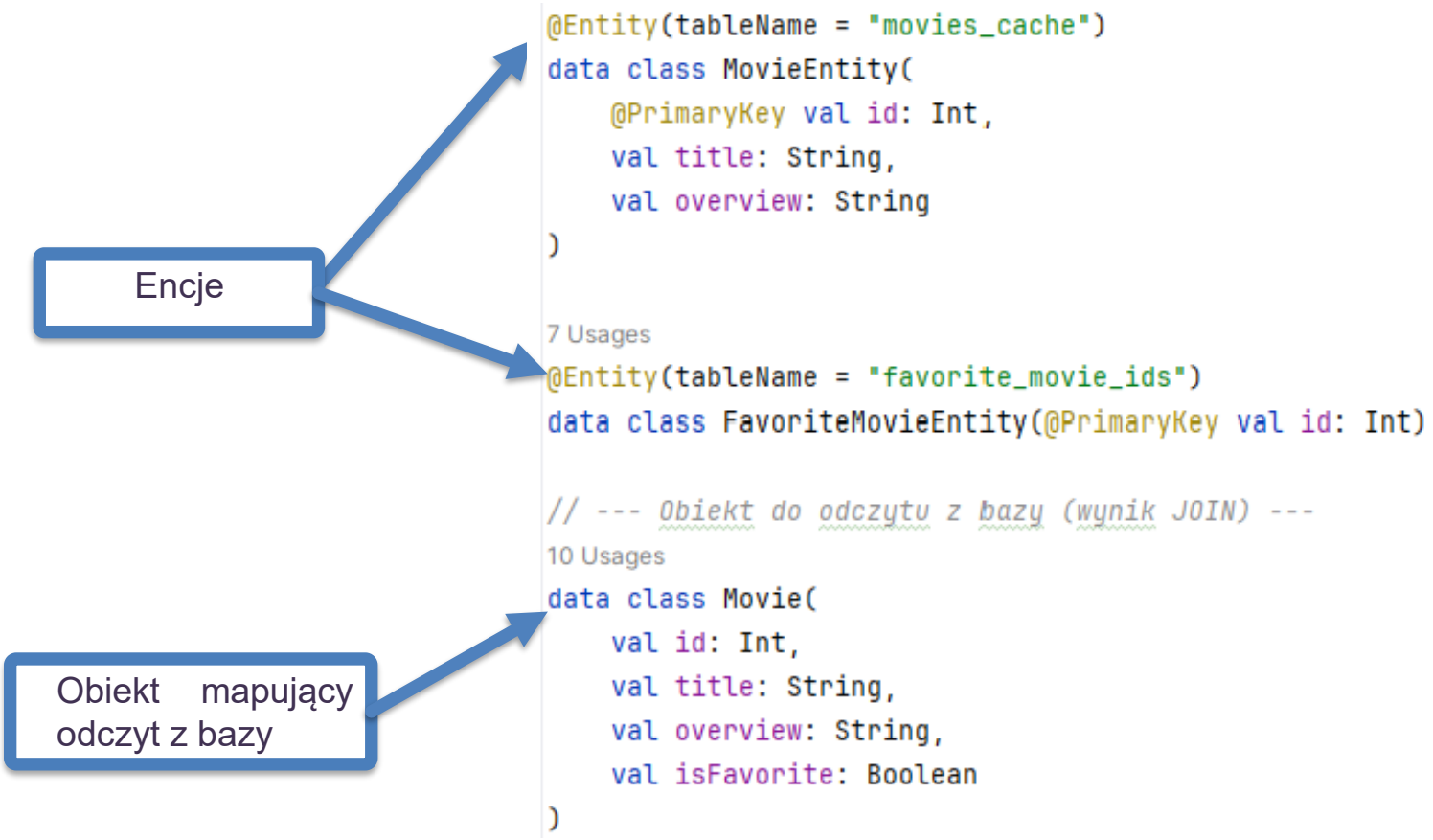
Kolejnym problemem jest trwałość stanu danych użytkownika (dodanie do ulubionych). Przy wielokrotnym odświeżaniu stan ten zostanie utracony. Tutaj jednym z rozwiązań jest zastosowanie ulotnego cache dla API oraz trwałego stanu preferencji użytkownika.

Najprostsze rozwiązanie: Wykorzystanie wielu tabel (tutaj dwóch) i łączenie danych.



Uniwersytet Wrocławski

Offline Caching




```kotlin
@Entity(tableName = "movies_cache")
data class MovieEntity(
    @PrimaryKey val id: Int,
    val title: String,
    val overview: String
)

7 Usages
@Entity(tableName = "favorite_movie_ids")
data class FavoriteMovieEntity(@PrimaryKey val id: Int)

// --- Obiekt do odczytu z bazy (wynik JOIN) ---
10 Usages
data class Movie(
    val id: Int,
    val title: String,
    val overview: String,
    val isFavorite: Boolean
)
```

Offline Caching

```
interface TmdbApiService {
    1 Usage
    @GET( value = "movie/popular")
    suspend fun getPopularMovies(
        @RetrofitQuery( value = "api_key") apiKey: String,
        @RetrofitQuery( value = "language") language: String = "pl-PL"
    ): MovieListResponse
}
```

W przypadku posiadania klas zawierających metody o **tej samej nazwie,** możemy nadać **alias** aby uniknąć konfliktów

```
import javax.inject.Singleton
import retrofit2.http.Query as RetrofitQuery
import androidx.room.Query
```

Uniwersytet Wrocławski

Offline Caching

Ta adnotacja zapewnia, że cała **operacja odczytu danych** z **wielu tabel** jest wykonana **atomowo**.

```
@Dao
interface MovieDao {
    1 Usage  1 Implementation
    @Transaction
    @Query( value = """
        SELECT
            movies_cache.*,
            (favorite_movie_ids.id IS NOT NULL) as isFavorite
        FROM movies_cache
        LEFT JOIN favorite_movie_ids ON movies_cache.id = favorite_movie_ids.id
        ORDER BY title ASC
    """)
    fun getMoviesWithFavoriteStatus(): Flow<List<Movie>>

    1 Usage  1 Implementation
    @Upsert
    suspend fun upsertMovies(movies: List<MovieEntity>)

    1 Implementation
    @Query( value = "DELETE FROM movies_cache")
    suspend fun clearMoviesCache()

    1 Usage  1 Implementation
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun addToFavorites(favorite: FavoriteMovieEntity)

    1 Usage  1 Implementation
    @Query( value = "DELETE FROM favorite_movie_ids WHERE id = :movieId")
    suspend fun removeFromFavorites(movieId: Int)
}
```

Uniwersytet Wrocławski

Offline Caching

Ta adnotacja zapewnia, że cała **operacja odczytu danych** z **wielu tabel** jest wykonana **atomowo**.

jakie kolumny chcemy pobrać

```
@Dao
interface MovieDao {
    1 Usage  1 Implementation
    @Transaction
    @Query( value = """
        SELECT
            movies_cache.*,
            (favorite_movie_ids.id IS NOT NULL) as isFavorite
        FROM movies_cache
        LEFT JOIN favorite_movie_ids ON movies_cache.id = favorite_movie_ids.id
        ORDER BY title ASC
    """)
    fun getMoviesWithFavoriteStatus(): Flow<List<Movie>>

    1 Usage  1 Implementation
    @Upsert
    suspend fun upsertMovies(movies: List<MovieEntity>)

    1 Implementation
    @Query( value = "DELETE FROM movies_cache")
    suspend fun clearMoviesCache()

    1 Usage  1 Implementation
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun addToFavorites(favorite: FavoriteMovieEntity)

    1 Usage  1 Implementation
    @Query( value = "DELETE FROM favorite_movie_ids WHERE id = :movieId")
    suspend fun removeFromFavorites(movieId: Int)
}
```

Uniwersytet Wrocławski

Offline Caching

Ta adnotacja zapewnia, że cała **operacja odczytu danych** z **wielu tabel** jest wykonana **atomowo**.

jakie kolumny chcemy pobrać

Wybierz wszystkie kolumny z tabeli movies_cache

```
@Dao
interface MovieDao {
    1 Usage  1 Implementation
    @Transaction
    @Query( value = """
        SELECT
            movies_cache.*,
            (favorite_movie_ids.id IS NOT NULL) as isFavorite
        FROM movies_cache
        LEFT JOIN favorite_movie_ids ON movies_cache.id = favorite_movie_ids.id
        ORDER BY title ASC
    """)
    fun getMoviesWithFavoriteStatus(): Flow<List<Movie>>

    1 Usage  1 Implementation
    @Upsert
    suspend fun upsertMovies(movies: List<MovieEntity>)

    1 Implementation
    @Query( value = "DELETE FROM movies_cache")
    suspend fun clearMoviesCache()

    1 Usage  1 Implementation
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun addToFavorites(favorite: FavoriteMovieEntity)

    1 Usage  1 Implementation
    @Query( value = "DELETE FROM favorite_movie_ids WHERE id = :movieId")
    suspend fun removeFromFavorites(movieId: Int)
}
```

Uniwersytet Wrocławski

Offline Caching

Ta adnotacja zapewnia, że cała **operacja odczytu danych** z **wielu tabel** jest wykonana **atomowo**.

jakie kolumny chcemy pobrać

Tworzy ona nową, **wirtualną** kolumnę o nazwie isFavorite

Wybierz wszystkie kolumny z tabeli movies_cache

```
@Dao
interface MovieDao {
    1 Usage  1 Implementation
    @Transaction
    @Query( value = """
        SELECT
            movies_cache.*,
            (favorite_movie_ids.id IS NOT NULL) as isFavorite
        FROM movies_cache
        LEFT JOIN favorite_movie_ids ON movies_cache.id = favorite_movie_ids.id
        ORDER BY title ASC
    """)
    fun getMoviesWithFavoriteStatus(): Flow<List<Movie>>

    1 Usage  1 Implementation
    @Upsert
    suspend fun upsertMovies(movies: List<MovieEntity>)

    1 Implementation
    @Query( value = "DELETE FROM movies_cache")
    suspend fun clearMoviesCache()

    1 Usage  1 Implementation
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun addToFavorites(favorite: FavoriteMovieEntity)

    1 Usage  1 Implementation
    @Query( value = "DELETE FROM favorite_movie_ids WHERE id = :movieId")
    suspend fun removeFromFavorites(movieId: Int)
}
```

Uniwersytet Wrocławski

Offline Caching

Ta adnotacja zapewnia, że cała **operacja odczytu danych** z **wielu tabel** jest wykonana **atomowo**.

jakie kolumny chcemy pobrać

Tworzy ona nową, **wirtualną** kolumnę o nazwie isFavorite

główna tabela, z której pobieramy dane

Wybierz wszystkie kolumny z tabeli movies_cache

```
@Dao
interface MovieDao {
    1 Usage  1 Implementation
    @Transaction
    @Query( value = """
        SELECT
            movies_cache.*,
            (favorite_movie_ids.id IS NOT NULL) as isFavorite
        FROM movies_cache
        LEFT JOIN favorite_movie_ids ON movies_cache.id = favorite_movie_ids.id
        ORDER BY title ASC
    """)
    fun getMoviesWithFavoriteStatus(): Flow<List<Movie>>

    1 Usage  1 Implementation
    @Upsert
    suspend fun upsertMovies(movies: List<MovieEntity>)

    1 Implementation
    @Query( value = "DELETE FROM movies_cache")
    suspend fun clearMoviesCache()

    1 Usage  1 Implementation
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun addToFavorites(favorite: FavoriteMovieEntity)

    1 Usage  1 Implementation
    @Query( value = "DELETE FROM favorite_movie_ids WHERE id = :movieId")
    suspend fun removeFromFavorites(movieId: Int)
}
```

Offline Caching

Ta adnotacja zapewnia, że cała **operacja odczytu danych** z **wielu tabel** jest wykonana **atomowo**.

jakie kolumny chcemy pobrać

Tworzy ona nową, **wirtualną** kolumnę o nazwie isFavorite

główna tabela, z której pobieramy dane

Weź każdy wiersz z **lewej tabeli** (movies_cache), a następnie spróbuj dołączyć do niego **pasujący wiersz z prawej tabeli** (favorite_movie_ids)

Wybierz wszystkie kolumny z tabeli movies_cache

```
@Dao
interface MovieDao {
    1 Usage  1 Implementation
    @Transaction
    @Query( value = """
        SELECT
            movies_cache.*,
            (favorite_movie_ids.id IS NOT NULL) as isFavorite
        FROM movies_cache
        LEFT JOIN favorite_movie_ids ON movies_cache.id = favorite_movie_ids.id
        ORDER BY title ASC
    """)
    fun getMoviesWithFavoriteStatus(): Flow<List<Movie>>

    1 Usage  1 Implementation
    @Upsert
    suspend fun upsertMovies(movies: List<MovieEntity>)

    1 Implementation
    @Query( value = "DELETE FROM movies_cache")
    suspend fun clearMoviesCache()

    1 Usage  1 Implementation
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun addToFavorites(favorite: FavoriteMovieEntity)

    1 Usage  1 Implementation
    @Query( value = "DELETE FROM favorite_movie_ids WHERE id = :movieId")
    suspend fun removeFromFavorites(movieId: Int)
}
```

Uniwersytet Wrocławski

Offline Caching

Ta adnotacja zapewnia, że cała **operacja odczytu danych** z **wielu tabel** jest wykonana **atomowo**.

jakie kolumny chcemy pobrać

Tworzy ona nową, **wirtualną** kolumnę o nazwie isFavorite

główna tabela, z której pobieramy dane

Weź każdy wiersz z **lewej tabeli** (movies_cache), a następnie spróbuj dołączyć do niego **pasujący wiersz z prawej tabeli** (favorite_movie_ids)

Jeśli dla jakiegoś filmu z movies_cache nie ma pasującego wpisu w favorite_movie_ids, wiersz ten wciąż zostanie zwrócony, a kolumny z favorite_movie_ids będą miały wartość NULL.

Wybierz wszystkie kolumny z tabeli movies_cache

```
@Dao
interface MovieDao {
    1 Usage  1 Implementation
    @Transaction
    @Query( value = """
        SELECT
            movies_cache.*,
            (favorite_movie_ids.id IS NOT NULL) as isFavorite
        FROM movies_cache
        LEFT JOIN favorite_movie_ids ON movies_cache.id = favorite_movie_ids.id
        ORDER BY title ASC
    """)
    fun getMoviesWithFavoriteStatus(): Flow<List<Movie>>

    1 Usage  1 Implementation
    @Upsert
    suspend fun upsertMovies(movies: List<MovieEntity>)

    1 Implementation
    @Query( value = "DELETE FROM movies_cache")
    suspend fun clearMoviesCache()

    1 Usage  1 Implementation
    @Insert(onConflict = OnConflictStrategy.IGNORE)
    suspend fun addToFavorites(favorite: FavoriteMovieEntity)

    1 Usage  1 Implementation
    @Query( value = "DELETE FROM favorite_movie_ids WHERE id = :movieId")
    suspend fun removeFromFavorites(movieId: Int)
```

Mutex

Użycie Mutex w Repository zapobiega konfliktom (race conditions) podczas **jednoczesnych prób** zapisu do bazy danych (np. odświeżanie i dodawanie do ulubionych w tym samym czasie).

```
class MovieRepository @Inject constructor(
    private val apiService: TmdbApiService,
    private val movieDao: MovieDao
) {
    1 Usage
    private val apiKey = "TUTAJ_WSTAW_SWOJ_KLUCZ_API"

    2 Usages
    private val mutex = Mutex()

    1 Usage
    val moviesStream: Flow<List<Movie>> = movieDao.getMoviesWithFavoriteStatus()

    1 Usage
    suspend fun toggleFavoriteStatus(movie: Movie) {
        mutex.withLock {
            if (movie.isFavorite) {
                movieDao.removeFromFavorites( movieId = movie.id)
            } else {
                movieDao.addToFavorites(FavoriteMovieEntity(id = movie.id))
            }
        }
    }
```

mutex.withLock to mechanizm, który pozwala **tylko jednemu wątkowi** lub **korutynie** na raz wykonać określony **fragment kodu.**

Uniwersytet Wrocławski

Wzorzec Aktor

Architektura ze wzorcem *Aktor* rozwiązuje problemy współbieżności, tworząc *kolejkę* dla operacji zapisu i **gwarantując**, że **intencje użytkownika nigdy nie zostaną utracone**.

Klasa sealed, która definiuje **wszystkie możliwe operacje** zapisu (Odśwież, Zmień Status Ulubionego).

```
sealed class DataAction {
    2 Usages
    data object RefreshMovies : DataAction()
    2 Usages
    data class ToggleFavorite(val movie: Movie) : DataAction()
}
```

Uniwersytet Wrocławski

Wzorzec Aktor

Repozytorium musi być singletonem, aby jeden aktor działał przez cały cykl życia aplikacji

```
@Singleton
class MovieRepository @Inject constructor(
    private val apiService: TmdbApiService,
    private val movieDao: MovieDao
) {
    1 Usage
    private val apiKey = "TUTAJ_WSTAW_SWOJ_KLUCZ_API"
    2 Usages
    private val actionChannel = Channel<DataAction>( capacity = Channel.UNLIMITED)
    1 Usage
    val moviesStream: Flow<List<Movie>> = movieDao.getMoviesWithFavoriteStatus()

    init {
        CoroutineScope( context = Dispatchers.IO).launch {
            for (action in actionChannel) {
                when (action) {
                    is DataAction.RefreshMovies -> performRefresh()
                    is DataAction.ToggleFavorite -> performToggleFavorite( movie = action.movie)
                }
            }
        }
    }
    2 Usages
    fun submitAction(action: DataAction) {
        actionChannel.trySend( element = action)
    }
    1 Usage
    private suspend fun performToggleFavorite(movie: Movie) {
        if (movie.isFavorite) {
            movieDao.removeFromFavorites( movieId = movie.id)
        } else {
            movieDao.addToFavorites(FavoriteMovieEntity(id = movie.id))
        }
    }
```

Uniwersytet Wrocławski

Wzorzec Aktor

Repozytorium musi być singletonem, aby jeden aktor działał przez cały cykl życia aplikacji

Prywatny kanał działający **jako kolejka** dla DataAction

```
@Singleton
class MovieRepository @Inject constructor(
    private val apiService: TmdbApiService,
    private val movieDao: MovieDao
) {
    1 Usage
    private val apiKey = "TUTAJ_WSTAW_SWOJ_KLUCZ_API"
    2 Usages
    private val actionChannel = Channel<DataAction>( capacity = Channel.UNLIMITED)
    1 Usage
    val moviesStream: Flow<List<Movie>> = movieDao.getMoviesWithFavoriteStatus()

    init {
        CoroutineScope( context = Dispatchers.IO).launch {
            for (action in actionChannel) {
                when (action) {
                    is DataAction.RefreshMovies -> performRefresh()
                    is DataAction.ToggleFavorite -> performToggleFavorite( movie = action.movie)
                }
            }
        }
    }
    2 Usages
    fun submitAction(action: DataAction) {
        actionChannel.trySend( element = action)
    }
    1 Usage
    private suspend fun performToggleFavorite(movie: Movie) {
        if (movie.isFavorite) {
            movieDao.removeFromFavorites( movieId = movie.id)
        } else {
            movieDao.addToFavorites(FavoriteMovieEntity(id = movie.id))
        }
    }
```

Uniwersytet Wrocławski

Wzorzec Aktor

Repozytorium musi być singletonem, aby jeden aktor działał przez cały cykl życia aplikacji

Prywatny kanał działający **jako kolejka** dla DataAction

Ta pętla **nasłuchuje** na nowe zadania w kanale.

```
@Singleton
class MovieRepository @Inject constructor(
    private val apiService: TmdbApiService,
    private val movieDao: MovieDao
) {
    1 Usage
    private val apiKey = "TUTAJ_WSTAW_SWOJ_KLUCZ_API"
    2 Usages
    private val actionChannel = Channel<DataAction>( capacity = Channel.UNLIMITED)
    1 Usage
    val moviesStream: Flow<List<Movie>> = movieDao.getMoviesWithFavoriteStatus()

    init {
        CoroutineScope( context = Dispatchers.IO).launch {
            for (action in actionChannel) {
                when (action) {
                    is DataAction.RefreshMovies -> performRefresh()
                    is DataAction.ToggleFavorite -> performToggleFavorite( movie = action.movie)
                }
            }
        }
    }
    2 Usages
    fun submitAction(action: DataAction) {
        actionChannel.trySend( element = action)
    }
    1 Usage
    private suspend fun performToggleFavorite(movie: Movie) {
        if (movie.isFavorite) {
            movieDao.removeFromFavorites( movieId = movie.id)
        } else {
            movieDao.addToFavorites(FavoriteMovieEntity(id = movie.id))
        }
    }
```

Uniwersytet Wrocławski

Wzorzec Aktor

Repozytorium musi być singletonem, aby jeden aktor działał przez cały cykl życia aplikacji

Prywatny kanał działający **jako kolejka** dla DataAction

Ta pętla **nasłuchuje** na nowe zadania w kanale.

Aktor **odbiera zadanie** i decyduje, którą z funkcji (performRefresh czy performToggleFavorite) wykonać

```
@Singleton
class MovieRepository @Inject constructor(
    private val apiService: TmdbApiService,
    private val movieDao: MovieDao
) {
    1 Usage
    private val apiKey = "TUTAJ_WSTAW_SWOJ_KLUCZ_API"
    2 Usages
    private val actionChannel = Channel<DataAction>( capacity = Channel.UNLIMITED)
    1 Usage
    val moviesStream: Flow<List<Movie>> = movieDao.getMoviesWithFavoriteStatus()

    init {
        CoroutineScope( context = Dispatchers.IO).launch {
            for (action in actionChannel) {
                when (action) {
                    is DataAction.RefreshMovies -> performRefresh()
                    is DataAction.ToggleFavorite -> performToggleFavorite( movie = action.movie)
                }
            }
        }
    }
    2 Usages
    fun submitAction(action: DataAction) {
        actionChannel.trySend( element = action)
    }
    1 Usage
    private suspend fun performToggleFavorite(movie: Movie) {
        if (movie.isFavorite) {
            movieDao.removeFromFavorites( movieId = movie.id)
        } else {
            movieDao.addToFavorites(FavoriteMovieEntity(id = movie.id))
        }
    }
```

Uniwersytet Wrocławski

Wzorzec Aktor

Repozytorium musi być singletonem, aby jeden aktor działał przez cały cykl życia aplikacji

Prywatny kanał działający **jako kolejka** dla DataAction

Ta pętla **nasłuchuje** na nowe zadania w kanale.

Aktor **odbiera zadanie** i decyduje, którą z funkcji (performRefresh czy performToggleFavorite) wykonać

To jedyna publiczna metoda do inicjowania zmian. ViewModel używa jej, aby dodać nowe zadanie do kolejki. trySend jest używane, aby operacja była **szybka i nieblokująca**

```
@Singleton
class MovieRepository @Inject constructor(
    private val apiService: TmdbApiService,
    private val movieDao: MovieDao
) {
    1 Usage
    private val apiKey = "TUTAJ_WSTAW_SWOJ_KLUCZ_API"
    2 Usages
    private val actionChannel = Channel<DataAction>( capacity = Channel.UNLIMITED)
    1 Usage
    val moviesStream: Flow<List<Movie>> = movieDao.getMoviesWithFavoriteStatus()

    init {
        CoroutineScope( context = Dispatchers.IO).launch {
            for (action in actionChannel) {
                when (action) {
                    is DataAction.RefreshMovies -> performRefresh()
                    is DataAction.ToggleFavorite -> performToggleFavorite( movie = action.movie)
                }
            }
        }
    }
    2 Usages
    fun submitAction(action: DataAction) {
        actionChannel.trySend( element = action)
    }
    1 Usage
    private suspend fun performToggleFavorite(movie: Movie) {
        if (movie.isFavorite) {
            movieDao.removeFromFavorites( movieId = movie.id)
        } else {
            movieDao.addToFavorites(FavoriteMovieEntity(id = movie.id))
        }
    }
```

Uniwersytet Wrocławski

Wzorzec Aktor

```
@HiltViewModel
class MoviesViewModel @Inject constructor(
    private val repository: MovieRepository
) : ViewModel() {

    1 Usage
    val uiState: StateFlow<MoviesUiState> = repository.moviesStream
        .map { movies -> MoviesUiState(movies = movies) }
        .stateIn(...)

    init {...}

    2 Usages
    fun refresh() {
        repository.submitAction( action = DataAction.RefreshMovies)
    }
    1 Usage
    fun onFavoriteClicked(movie: Movie) {
        repository.submitAction( action = DataAction.ToggleFavorite(movie))
    }
}
```

ViewModel wywołuje tylko submitAction z Repository

Podsumowanie

- **Problem do Rozwiązania**: Aplikacje zależne wyłącznie od połączenia z siecią (Retrofit) przestają działać w **trybie offline**, co prowadzi do występowania błędów (pusty ekran, błąd).
- **Wzorzec "Single Source of Truth" (SSoT)**: To kluczowa zasada, która mówi, że interfejs użytkownika (UI) powinien czerpać dane wyłącznie z **jednego, lokalnego źródła prawdy** – w naszym przypadku z bazy danych Room.
- **Dwa Główne Przepływy Danych:**
  - **Ścieżka Odczytu:** UI reaktywnie obserwuje Flow wystawiony przez Room. Zapewnia to **natychmiastowe wyświetlanie danych** (nawet starych) i **automatyczne odświeżanie** po zmianie w bazie.
  - **Ścieżka Zapisu:** Repository w tle pobiera aktualne dane z sieci (Retrofit), a następnie **aktualizuje** nimi źródło prawdy (zapisuje je do Room), co pośrednio powoduje odświeżenie UI.
- **Zachowanie Stanu Użytkownika:** Aby zachować **lokalne zmiany** (np. status *ulubione*) podczas odświeżania z sieci, stosuje się zaawansowane techniki, takie jak **osobna tabela na preferencje użytkownika** i łączenie jej z danymi z cache'u za pomocą zapytania JOIN w DAO.
- **Obsługa Współbieżności:** Aby **uniknąć konfliktów** przy **jednoczesnym zapisie do bazy** (np. odświeżanie i zmiana statusu *ulubione*), używa się **mechanizmów synchronizacji**, takich jak **Mutex** lub **Wzorzec Aktor z Channel**, które zapewniają, że **operacje zapisu** wykonują się **jedna po drugiej.**