



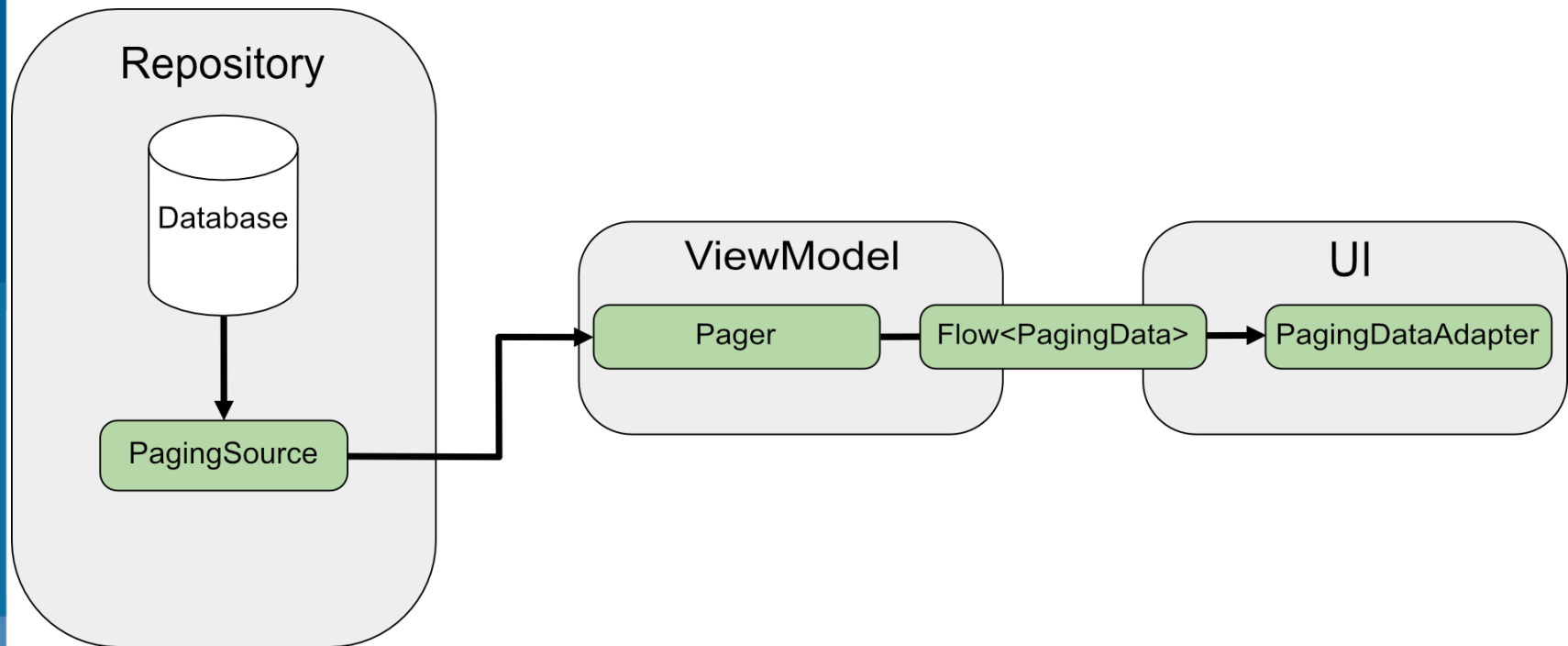
Uniwersytet
Wrocławski

Rafał Lewandków

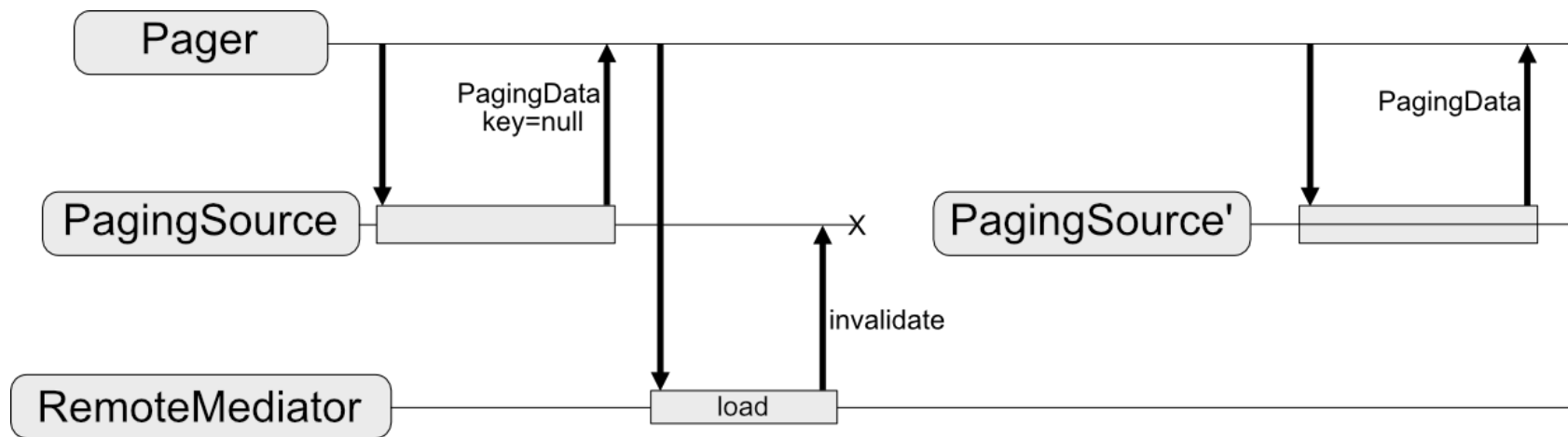
PROGRAMOWANIE URZĄDZEŃ MOBILNYCH

WYKŁAD 13
Paging

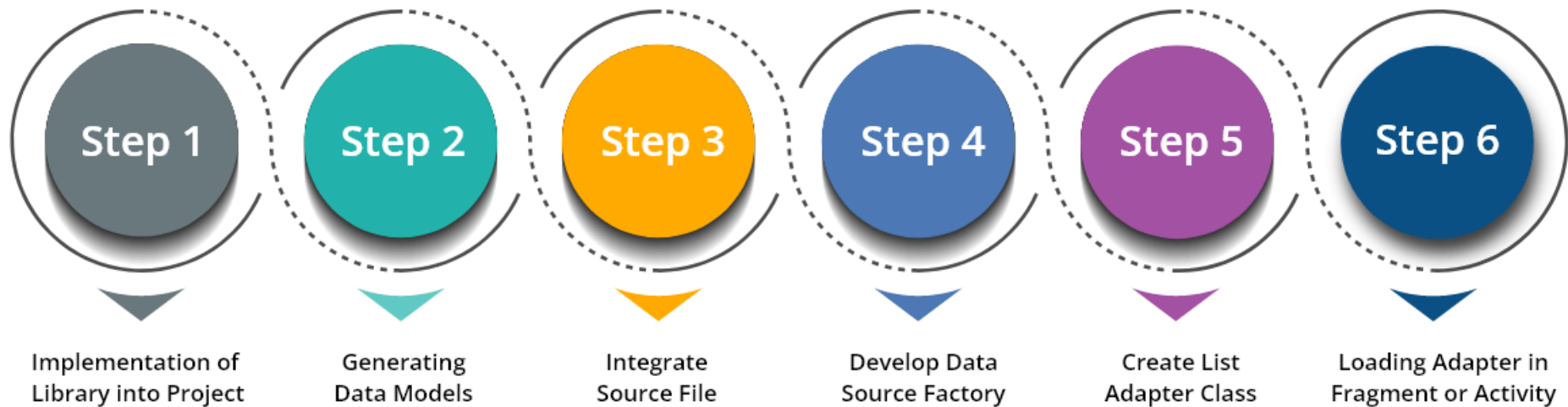
Paging



Paging



Implement the Paging 3 Library in Android Kotlin



<https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.oneclicksolution.com%2Fblog%2Fpaging-3-library-in-android%2F>

Biblioteka `Paging3` to narzędzie do paginacji, które ułatwia ładowanie danych partiami w aplikacjach.

- Efektywne ładowanie danych: Biblioteka pomaga w efektywnym zarządzaniu ładowaniem dużych zbiorów danych, takich jak listy elementów w aplikacjach z wieloma rekordami. Dzięki niemu możesz ładować dane partiami, co pozwala na lepszą wydajność i oszczędność zasobów urządzenia.
- Aktualizacje w czasie rzeczywistym: Biblioteka umożliwia automatyczne monitorowanie źródła danych i dostarcza aktualizacje w czasie rzeczywistym, na przykład w odpowiedzi na zmiany w bazie danych lub na serwerze.
- Łatwe obsługiwanie błędów i odzyskiwanie: Biblioteka zawiera wbudowane mechanizmy obsługi błędów i odzyskiwania, co ułatwia zarządzanie sytuacjami takimi jak utrata połączenia internetowego lub błędy zapytań do źródła danych.
- Obsługa różnych źródeł danych: Możesz używać biblioteki z różnymi źródłami danych, takimi jak baza danych lokalna, zapytania sieciowe lub inne źródła danych.
- Integracja z interfejsem użytkownika: Biblioteka jest łatwa do zintegrowania z interfejsem użytkownika. Pozwala na płynne przewijanie i dynamiczne ładowanie nowych danych w miarę przewijania listy.
- Aby korzystać z biblioteki, musisz zdefiniować źródło danych `PagingSource`, które dostarcza dane `PagingData`. Biblioteka automatycznie zarządza ładowaniem danych w odpowiednich momentach, co ułatwia tworzenie wydajnych i responsywnych aplikacji Android.
- Biblioteka posiada wbudowane mechanizmy obsługi stanów (`LoadingState.Error`, `LoadingState.Append` etc.)

```
{
  "count": 82,
  "next": "https://swapi.dev/api/people/?page=2",
  "previous": null,
  "results": [
    {
      "name": "Luke Skywalker",
      "height": "172",
      "mass": "77",
      "hair_color": "blond",
      "skin_color": "fair",
      "eye_color": "blue",
      "birth_year": "19BBY",
      "gender": "male",
      "homeworld": "https://swapi.dev/api/planets/1/",
      "films": [
        "https://swapi.dev/api/films/1/",
        "https://swapi.dev/api/films/2/",
        "https://swapi.dev/api/films/3/",
        "https://swapi.dev/api/films/6/"
      ],
      "species": [],
      "vehicles": [
        "https://swapi.dev/api/vehicles/14/",
        "https://swapi.dev/api/vehicles/30/"
      ],
      "starships": [
        "https://swapi.dev/api/starships/12/",
        "https://swapi.dev/api/starships/22/"
      ],
      "created": "2014-12-09T13:50:51.644000Z",
      "edited": "2014-12-20T21:17:56.891000Z",
      "url": "https://swapi.dev/api/people/1/"
    },
    ...
  ]
}
```

```
data class SwapResponse(  
    val count: Int,  
    val next: String?,  
    val previous: String?,  
    val results: List<Result>  
)
```

```
data class Result(  
    val height: String,  
    @SerializedName("homeworld") val homeWorld: String,  
    val name: String,  
)
```

```
data class SwapiResponse(  
    val count: Int,  
    val next: String?,  
    val previous: String?,  
    val results: List<Result>  
)
```

```
data class Result(  
    val height: String,  
    @SerializedName("homeworld") val homeworld: String,  
    val name: String,  
)
```

```
interface SwapiApiService {  
    @GET("people")  
    suspend fun getCharacters(  
        @Query("page") page: Int  
    ): SwapiResponse  
}
```

```
object RetrofitInstance {  
    val api: SwapiApiService by lazy {  
        Retrofit.Builder()  
            .baseUrl("https://swapi.dev/api/")  
            .addConverterFactory(GsonConverterFactory.create())  
            .build()  
            .create(SwapiApiService::class.java)  
    }  
}
```



```
data class SwapiResponse(  
    val count: Int,  
    val next: String?,  
    val previous: String?,  
    val results: List<Result>  
)
```

```
data class Result(  
    val height: String,  
    @SerializedName("homeworld") val homeworld: String,  
    val name: String,  
)
```

```
interface SwapiApiService {  
    @GET("people")  
    suspend fun getCharacters(  
        @Query("page") page: Int  
    ): SwapiResponse  
}
```

```
object RetrofitInstance {  
    val api: SwapiApiService by lazy {  
        Retrofit.Builder()  
            .baseUrl("https://swapi.dev/api/")  
            .addConverterFactory(GsonConverterFactory.create())  
            .build()  
            .create(SwapiApiService::class.java)  
    }  
}
```

```
class SwapiRepository {  
    private val api = RetrofitInstance.api  
  
    suspend fun getCharacters(page: Int) = api.getCharacters(page)  
}
```

```
class CharactersPagingSource(  
    private val repository: SwapiRepository  
) : PagingSource<Int, Result>() {  
    override fun getRefreshKey(state: PagingState<Int, Result>): Int? {  
        return state.anchorPosition ?.let { anchorPosition ->  
            val anchorPage = state.closestPageToPosition(anchorPosition)  
            anchorPage?.prevKey?.plus(1) ?: anchorPage?.nextKey?.minus(1)  
        }  
    }  
  
    override suspend fun load(params: LoadParams<Int>): LoadResult<Int, Result> {  
        return try {  
            val page = params.key ?: 1  
            val response = repository.getCharacters(page)  
  
            LoadResult.Page(  
                data = response.results,  
                prevKey = getPageNumberFromUrl(response.previous),  
                nextKey = getPageNumberFromUrl(response.next),  
            )  
        } catch (e: Exception) {  
            LoadResult.Error(e)  
        }  
    }  
  
    private fun getPageNumberFromUrl(url: String?): Int? {  
        if (url != null) {  
            val pattern = Pattern.compile("page=(\\d+)")  
            val matcher = pattern.matcher(url)  
  
            if (matcher.find()) {  
                val pageNumberString = matcher.group(1)  
                if (pageNumberString != null) {  
                    return pageNumberString.toInt()  
                }  
            }  
        }  
  
        return null  
    }  
}
```

```
override fun getRefreshKey(state: PagingState<Int, Result>): Int? {  
    return state.anchorPosition ?.let { anchorPosition ->  
        val anchorPage = state.closestPageToPosition(anchorPosition)  
        anchorPage?.prevKey?.plus(1) ?: anchorPage?.nextKey?.minus(1)  
    }  
}
```

- `state.anchorPosition` : To pozycja *kotwicy* (anchor position) w stanie paginacji (`PagingState`). Pozycja kotwicy to pozycja w widoku, który jest uważany za aktualnie wyświetlany przez użytkownika, np. pozycja pierwszego widocznego elementu na liście.
- `?.let { anchorPosition -> ... }` : Jest to blok, który wykonuje kod wewnątrz tylko wtedy, gdy `state.anchorPosition` nie jest `null` . Wartość `anchorPosition` jest dostępna wewnątrz tego bloku.
- `val anchorPage = state.closestPageToPosition(anchorPosition)` : Ta linia kodu pobiera stronę (`page`) najbliższą pozycji kotwicy. Strona w kontekście paginacji to fragment danych, który jest ładowany partiami.
- `anchorPage?.prevKey?.plus(1) ?: anchorPage?.nextKey?.minus(1)` : Ta linia kodu określa klucz odświeżania. Jeśli możliwy jest klucz poprzedniej strony (`prevKey`) dla strony kotwicy, to jest on inkrementowany o 1 (czyli następna strona). W przeciwnym razie, jeśli możliwy jest klucz następnej strony (`nextKey`) dla strony kotwicy, to jest on dekrementowany o 1 (czyli poprzednia strona). Jeśli oba klucze są `null` , wynikiem jest `null` .

```
class SwapiViewModel : ViewModel() {  
    private val repository = SwapiRepository()  
  
    fun getData() = Pager(  
        config = PagingConfig(  
            pageSize = 10,  
        ),  
        pagingSourceFactory = {  
            CharactersPagingSource(repository)  
        }  
    ).flow.cachedIn(viewModelScope)  
}
```

- `fun getData() = Pager(...)`: Jest to funkcja, która zwraca strumień (`Flow`) paginowanych danych. Funkcja ta korzysta z biblioteki `Paging`, aby zarządzać paginacją danych.
- `Pager(...)` to konstruktor klasy `Pager`, który inicjuje mechanizm paginacji danych. Skonfigurowane są następujące parametry:
 - `config = PagingConfig(pageSize = 10)`: Konfiguracja paginacji zdefiniowana za pomocą `PagingConfig`. Określono, że rozmiar strony (`pageSize`) wynosi 10, co oznacza, że dane będą ładowane partiami po 10 elementów na stronę.
 - `pagingSourceFactory = { CharactersPagingSource(repository) }`: Właściwość `pagingSourceFactory` definiuje, jakie źródło danych (`CharactersPagingSource`) zostanie użyte do ładowania danych. Przekazano repozytorium jako argument do źródła danych.
- `.flow`: Po konfiguracji pagera, wywołujemy `.flow`, aby uzyskać strumień danych paginowanych, który może być obserwowany przez interfejs użytkownika.
- `.cachedIn(viewModelScope)`: Metoda `.cachedIn` zapewnia, że strumień danych będzie przechowywany w danym zakresie (`viewModelScope`), co jest ważne, aby uniknąć utraty danych w przypadku ponownego tworzenia `ViewModel` w trakcie cyklu życia aplikacji.

RecyclerView + Paging

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

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

```
class CharacterAdapter(characterComparator: CharacterComparator) : PagingDataAdapter<Result, CharacterViewHolder>(characterComparator) {  
    override fun onBindViewHolder(holder: CharacterViewHolder, position: Int) {  
        val item = getItem(position) ?: Result("", "", "")  
        holder.bind(item)  
    }  
  
    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CharacterViewHolder {  
        return CharacterViewHolder(  
            RvItemBinding.inflate(  
                LayoutInflater.from(parent.context), parent, false  
            )  
        )  
    }  
}
```

```
val characterAdapter = CharacterAdapter(CharacterComparator())

viewLifecycleOwner.lifecycleScope.launch {
    viewModel.getData().collectLatest { pagingData ->
        characterAdapter.submitData(pagingData)
    }
}

binding.recycler.apply {
    layoutManager = LinearLayoutManager(requireContext())
    adapter = characterAdapter
    setHasFixedSize(true)
}
```

Paging + Compose

```
@Composable
fun ListScreen(){
    val viewModel: SwapiViewModel = viewModel()

    val characters = viewModel.getData().collectAsLazyPagingItems()

    LazyColumn(
        modifier = Modifier.fillMaxSize(),
        horizontalAlignment = Alignment.CenterHorizontally
    ) {
        items(characters.itemCount){
            characters[it]?.let { item ->
                Text(
                    text = item.name,
                    textAlign = TextAlign.Center,
                    fontSize = 24.sp
                )
            }
        }
    }
}
```

Funkcja `collectAsLazyPagingItems` jest częścią biblioteki `Paging` w połączeniu z biblioteką `Compose`. Ta funkcja służy do przekształcania strumienia danych paginowanych na listę elementów, którą można wygodnie używać w interfejsie użytkownika stworzonym w `Compose`.

- Pobiera strumień danych paginowanych: Na wejściu `collectAsLazyPagingItems` przyjmuje strumień danych paginowanych. Ten strumień reprezentuje dane, które są ładowane partiami, co jest typowe w przypadku list lub zestawów danych o dużej ilości elementów.
- Przekształca strumień na listę elementów: Funkcja przekształca ten strumień w listę, która jest leniwa (`lazy`). Oznacza to, że nie wszystkie dane są ładowane od razu, ale tylko wtedy, gdy są potrzebne. Dzięki temu można skutecznie obsługiwać duże zbiory danych, ładować tylko część z nich na raz i oszczędzać zasoby systemu.
- Umożliwia wygodne korzystanie z danych: Po przekształceniu strumienia na listę, można wygodnie iterować przez elementy, wyświetlać je w interfejsie użytkownika lub wykonywać na nich inne operacje, takie jak filtrowanie czy sortowanie.