ANDROID 254

01

The World of Paging

MAMBO BRYAN
Android Developer | NLS Tech Solutions

- What is paging?
- Why do we need it?
- How can we page our data?
- What are the benefits?

02

What is our goal?

Today's Agenda

03

Paging? Pagination?

Pagination is fetching information incrementally from a large dataset.

So what is paging then?

04

So what is Paging really?

An awesome library from google.

Whilst pagination is fetching data incrementally, the paging library helps us to implement pagination quickly and gracefully.

The paging library is like a Matatu Conductor.

05

Why do we page data?

Because we can! No seriously, why?

"Just because it works doesn't mean it's usable."
- YouKnowYourself

06

Understanding the Problems

Limited System Resources

Unknown Network Connection

User Experience Needs

Server costs

07

This is the Way!

Room

How to get paged data from the local database with Room.

Retrofit

Getting paged data from a remote server with Retrofit without cache

Firestore

Getting Firebase firestore NoSql database

Room & Retrofit

Getting paged data from a remote server with Retrofit and caching it to the local database

How should it look like?

08 This is important...

Paging Source

The base class for loading chunks of data for a specific page query.

Remote Mediator

The base class for loading chunks of data from a server and saving it to a local database.

Pager

A class responsible for producing the PagingData stream and depends on the PagingSource.

Paging Data

A container for paginated data. Where the paged list is contain

09

Room (Local Database)

Feel free to make this an open discussion for questions or clarifications.

Entity →

```kotlin
/**
 * 1.1 -> Create an Entity Class
 */
@Entity(tableName = "notes")
data class Note(
    @PrimaryKey(autoGenerate = true) var id: Int = 0,
    var description: String? = ""
)
```

```kotlin
/**
 * 1.2 -> Create your entity's Data Access Object (DAO)
 */
@Dao
interface NotesDao {

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(note:Note)

    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insert(notes: List<Note>)

    @Query("DELETE FROM notes WHERE id IS :id")
    suspend fun delete(id: Int)

    @Query("DELETE FROM notes")
    suspend fun deleteAll()

    /**
     * 1.3 -> Notice that Room can return a PagingSource, isn't that amazing?
     */
    @Query("SELECT * FROM notes")
    fun getAllNotes(): PagingSource<Int, Note>

}
```

← Dao

```
/**
 * 1.4 -> Create a repository
 */

class NoteRepository @Inject constructor(
    private val database: AppDatabase
) {
    /**
     * 1.4 -> Create a pager
     */

    fun getRoomNotes() = Pager(
        config = PagingConfig(pageSize = 20, enablePlaceholders = false),
        pagingSourceFactory = { database.notesDao().getAllNotes() }
    ).flow

}
```

← Repository

ViewModel →

```
@HiltViewModel
class NotesViewModel @Inject constructor(
    repository: NoteRepository
) : ViewModel() {

    /**
     * 1.5 -> Get all notes from repository
     */
    val roomNotes: Flow<PagingData<Note>> =
        repository.getRoomNotes().cachedIn(viewModelScope)

}
```

```
@AndroidEntryPoint
class RoomFragment : Fragment(R.layout.fragment_template) {

    private val binding by viewBinding(FragmentTemplateBinding::bind)
    private val viewModel: NotesViewModel by activityViewModels()

    @Inject
    lateinit var noteAdapter: NoteAdapter

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        initViews()

        /**
         * 1.7 -> Add load state adapter to the Main adapter
         */
        lifecycleScope.launch {
            noteAdapter.loadStateFlow.collectLatest { loadState ->

                val isListEmpty =
                    loadState.refresh is LoadState.NotLoading && noteAdapter.itemCount == 0
                // Show empty state if list is Empty
                binding.layoutState.stateEmpty.isVisible = isListEmpty
                // Only Show the content if refresh succeeds.
                binding.layoutState.stateContent.isVisible = !isListEmpty
                // Show loading state during initial load or refresh.
                binding.layoutState.stateLoading.isVisible =
                    loadState.source.refresh is LoadState.Loading
                // Show the error state if initial load or refresh fails.
                binding.layoutState.stateError.isVisible =
                    loadState.source.refresh is LoadState.Error
                // Get Error message from PagingSource
                val errorState = loadState.source.append as? LoadState.Error
                    ?: loadState.source.prepend as? LoadState.Error
                    ?: loadState.append as? LoadState.Error
                    ?: loadState.prepend as? LoadState.Error
                errorState?.let {
                    binding.layoutState.tvError.text = "\uD83D\uDE28 Wooops ${it.error}"
                }

            }
        }

        /**
         * 1.6 -> Collect latest paging data
         */
        lifecycleScope.launch {
            viewModel.roomNotes.collectLatest {
                noteAdapter.submitData(it)
            }
        }

    }

    private fun initViews() {

        binding.layoutState.tvError.text = "Couldn't Load Notes"
        binding.layoutState.tvEmpty.text = "No Note Found"
        binding.layoutState.buttonRetry.setOnClickListener { noteAdapter.refresh() }

        binding.layoutState.recyclerView.apply {
            setHasFixedSize(true)
            adapter = noteAdapter
        }

    }

}
```

UI

Load State

Handles success, failure and empty

10

Retrofit (Remote Database)

Getting character using the Rick & Morty API.

Sample Response →

```
/**
 * 2.2 -> Create a Character class with the values you need
 */
data class Character(

    @SerializedName("id") var id: Int? = null,
    @SerializedName("name") var name: String? = null,
    @SerializedName("status") var status: String? = null,
    @SerializedName("species") var species: String? = null,
    @SerializedName("type") var type: String? = null,
    @SerializedName("gender") var gender: String? = null,
    @SerializedName("image") var image: String? = null,
    @SerializedName("url") var url: String? = null,
    @SerializedName("created") var created: String? = null

)

data class Origin(

    @SerializedName("name") var name: String? = null,
    @SerializedName("url") var url: String? = null

)

data class Location(

    @SerializedName("name") var name: String? = null,
    @SerializedName("url") var url: String? = null

)
```

```
/**
 * 2.1 -> Create a Response class
 */

data class CharacterResponse(

    @SerializedName("info") var info: ResponseInfo? = ResponseInfo(),
    @SerializedName("results") var characters: ArrayList<Character> = arrayListOf()

)

data class ResponseInfo(

    @SerializedName("count") var count: Int? = null,
    @SerializedName("pages") var pages: Int? = null,
    @SerializedName("next") var next: String? = null,
    @SerializedName("prev") var prev: String? = null

)
```

← Model

```
/**
 * 2.3 -> Setup your API service to return the custom response class
 */
interface ApiService {


    @GET("character")
    suspend fun getCharacters(@Query("page") page: Int):
                CharacterResponse


}
```

ApiService ↑

Paging Source →

```
/**
 * 2.4 -> Create a custom paging source class and extend the paging library
 *        PagingSource class
 */
class CharactersPagingSource(
    private val apiService: ApiService,
) : PagingSource<Int, Character>() {

    override fun getRefreshKey(state: PagingState<Int, Character>): Int? {
        // We need to get the previous key (or next key if previous is null) of the page
        // that was closest to the most recently accessed index.
        // Anchor position is the most recently accessed index
        return state.anchorPosition?.let { anchorPosition ->
            state.closestPageToPosition(anchorPosition)?.prevKey?.plus(1)
                ?: state.closestPageToPosition(anchorPosition)?.nextKey?.minus(1)
        }
    }

    override suspend fun load(params: LoadParams<Int>): LoadResult<Int, Character> {

        // Get the current page from the params
        val page = params.key ?: Constants.START_PAGE

        return try {
            // Get the response from the server
            val response = apiService.getCharacters(page)

            // Get the list of characters in the page
            val characters = response.characters

            // Get the next key for loading the next page
            val nextKey = if(response.info?.next.isNullOrBlank()) null else page + 1
            // val nextKey = if (articles.isEmpty()) null else page + 1

            // Get the previous key
            val previousKey = if (page == Constants.START_PAGE) null else page - 1

            delay(3000)

            // Return the LoadResult with characters, previousKey and nextKey
            LoadResult.Page(data = characters, prevKey = previousKey, nextKey = nextKey)

        } catch (exception: IOException) {
            return LoadResult.Error(exception)
        } catch (exception: HttpException) {
            return LoadResult.Error(exception)
        }

    }
}
```

```kotlin
/**
 * 2.5 -> Create a character repository passing the ApiService class
 */

class CharactersRepository @Inject constructor(
    private val apiService: ApiService
) {

    companion object {
        const val PAGE_SIZE = 20
    }

    /**
     * 2.6 -> Create a pager that return the paging data stream
     */
    fun getCharacters(): Flow<PagingData<Character>> {
        return Pager(
            config = PagingConfig(pageSize = PAGE_SIZE, enablePlaceholders = false),
            pagingSourceFactory = { CharactersPagingSource(apiService) }
        ).flow
    }

}
```

← Repository

ViewModel →

```kotlin
package com.mambobryan.samba.ui.characters

import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import androidx.paging.cachedIn
import com.mambobryan.samba.data.repositories.CharactersRepository
import dagger.hilt.android.lifecycle.HiltViewModel
import javax.inject.Inject

/**
 * 2.7 -> Create a viemodel class and get the characters
 */
@HiltViewModel
class CharactersViewModel @Inject constructor(
    repository: CharactersRepository
) : ViewModel(){

    val articles = repository.getCharacters().cachedIn(viewModelScope)

    val articlesWithCache = repository.getCharactersWithCache().cachedIn(viewModelScope)

}
```

```kotlin
@AndroidEntryPoint
class RemoteOnlyFragment : Fragment(R.layout.fragment_template) {

    private val binding by viewBinding(FragmentTemplateBinding::bind)
    private val viewModel: CharactersViewModel by activityViewModels()

    @Inject
    lateinit var characterAdapter: CharacterAdapter

    @Inject
    lateinit var footer: ItemLoadStateAdapter

    override fun onViewCreated(view: View, savedInstanceState: Bundle?) {
        super.onViewCreated(view, savedInstanceState)

        initViews()

        /**
         * 2.8 -> Handle the load states
         */
        lifecycleScope.launch {
            characterAdapter.loadStateFlow.collectLatest { loadState ->

                val isListEmpty =
                    loadState.refresh is LoadState.NotLoading && characterAdapter.itemCount == 0
                // show empty list
                binding.layoutState.stateEmpty.isVisible = isListEmpty
                // Only show the list if refresh succeeds.
                binding.layoutState.stateContent.isVisible = !isListEmpty
                // Show loading spinner during initial load or refresh.
                binding.layoutState.stateLoading.isVisible =
                    loadState.source.refresh is LoadState.Loading
                // Show the retry state if initial load or refresh fails.
                binding.layoutState.stateError.isVisible =
                    loadState.source.refresh is LoadState.Error
                // Show any error that comes from PagingSource
                val errorState = loadState.source.append as? LoadState.Error
                    ?: loadState.source.prepend as? LoadState.Error
                    ?: loadState.append as? LoadState.Error
                    ?: loadState.prepend as? LoadState.Error
                errorState?.let {
                    binding.layoutState.tvError.text = "Wooops ${it.error}"
                }

            }
        }

        /**
         * 2.7 -> Collect the list of characters and submit it to the adapter
         */
        lifecycleScope.launch {
            viewModel.articles.collectLatest {
                characterAdapter.submitData(it)
            }
        }

    }

    private fun initViews() {

        binding.layoutState.buttonRetry.setOnClickListener { characterAdapter.refresh() }

        /**
         * 2.9 -> Let's go further and add a footer
         */
        footer.onRetryClicked { characterAdapter.retry() }
        characterAdapter.withLoadStateFooter(footer = footer)

        binding.layoutState.recyclerView.apply {
            setHasFixedSize(true)
            adapter = characterAdapter
        }

    }

}
```

UI

Load State

Handles success, failure and empty

Load State Adapter

Add load state adapter

Load State Adapter

```
/**
 * 2.10 -> Create a load state adapter to show progress
 */

class ItemLoadStateAdapter @Inject constructor() :
    LoadStateAdapter<ItemLoadStateAdapter.ItemLoadStateViewHolder>() {

    private var retry: (() -> Unit)? = null

    fun onRetryClicked(retry: () -> Unit) {
        this.retry = retry
    }

    inner class ItemLoadStateViewHolder(private val binding: LayoutLoadStateBinding) :
        RecyclerView.ViewHolder(binding.root) {

        init {
            binding.btnStateRetry.setOnClickListener { retry?.invoke() }
        }

        fun bind(loadState: LoadState) {

            /**
             * 2.11 -> Handle request Loading and Error
             */

//            when(loadState){
//                LoadState.Loading -> TODO()
//                is LoadState.NotLoading -> TODO()
//                is LoadState.Error -> TODO()
//            }

            binding.apply {
                if (loadState is LoadState.Error) tvStateError.text =
                loadState.error.localizedMessage

                progressState.isVisible = loadState is LoadState.Loading
                tvStateError.isVisible = loadState is LoadState.Error
                btnStateRetry.isVisible = loadState is LoadState.Error

                if (loadState.endOfPaginationReached) {
                    tvStateError.text = "Loaded All Items"
                    tvStateError.isVisible = true
                    btnStateRetry.isVisible = false
                }

            }
        }

    }

    override fun onBindViewHolder(holder: ItemLoadStateAdapter.ItemLoadStateViewHolder,
                                  loadState: LoadState) {
        holder.bind(loadState)
    }

    override fun onCreateViewHolder(
        parent: ViewGroup,
        loadState: LoadState
    ): ItemLoadStateAdapter.ItemLoadStateViewHolder {
        val binding =
            LayoutLoadStateBinding.inflate(LayoutInflater.from(parent.context), parent, false)
        return ItemLoadStateViewHolder(binding)
    }

}
```

11

Firestore (Remote Database)

Getting a list of documents from Firestore NoSQL database.

Document Class

```kotlin
/**
 * 3.1 -> Add document id and ignore it in room
 */
@Entity(tableName = "notes")
data class Note(
    @PrimaryKey(autoGenerate = true) @Exclude var id: Int = 0,
    @DocumentId @Ignore var docId: String? = "",
    @Ignore var date: Date? = null,
    var description: String? = ""
)
```

```kotlin
/**
 * 3.2 -> Create a custom paging source class and extend the paging library
 *        PagingSource class
 */
class NotesPagingSource(
    private val collection: CollectionReference
) : PagingSource<DocumentSnapshot, Note>() {

    override fun getRefreshKey(state: PagingState<DocumentSnapshot, Note>): DocumentSnapshot? {
        return state.anchorPosition?.let { anchorPosition ->
            state.closestPageToPosition(anchorPosition)?.prevKey
                ?: state.closestPageToPosition(anchorPosition)?.nextKey
        }
    }

    override suspend fun load(params: LoadParams<DocumentSnapshot>): LoadResult<DocumentSnapshot, Note>
{

        val document = params.key

        return try {

            val query = collection.orderBy("date", Query.Direction.DESCENDING).limit(20)

            val documents =
                if (document == null) query.get().await()
                else query.startAfter(document).get().await()

            val nextKey = if (documents.isEmpty) null else documents.last()
//            val nextKey = documents.lastOrNull()

            val notes = documents.toObjects(Note::class.java)

            delay(3000)

            Timber.i("Loaded Items Size => ${documents.size()}")

            LoadResult.Page(data = notes, null, nextKey = nextKey)

        } catch (exception: FirebaseFirestoreException) {
            return LoadResult.Error(exception)
        }

    }
}
```

Paging Source

What about the key?

AOur key becomes a document snapshot

Repository

```
/**
 * 1.4 -> Create a repository
 */

class NoteRepository @Inject constructor(
    private val database: AppDatabase
) {

    private val collections = Firebase.firestore.collection("notes")

    /**
     * 1.4 -> Create a pager for local notes
     */

    fun getRoomNotes() = Pager(
        config = PagingConfig(pageSize = 20, enablePlaceholders = false),
        pagingSourceFactory = { database.notesDao().getAllNotes() }
    ).flow

    /**
     * 3.3 -> Create a pager for firestore notes
     */
    fun getFirestoreNotes() = Pager(
        config = PagingConfig(pageSize = 20, enablePlaceholders = false),
        pagingSourceFactory = { NotesPagingSource(collections) }
    ).flow

}
```

12

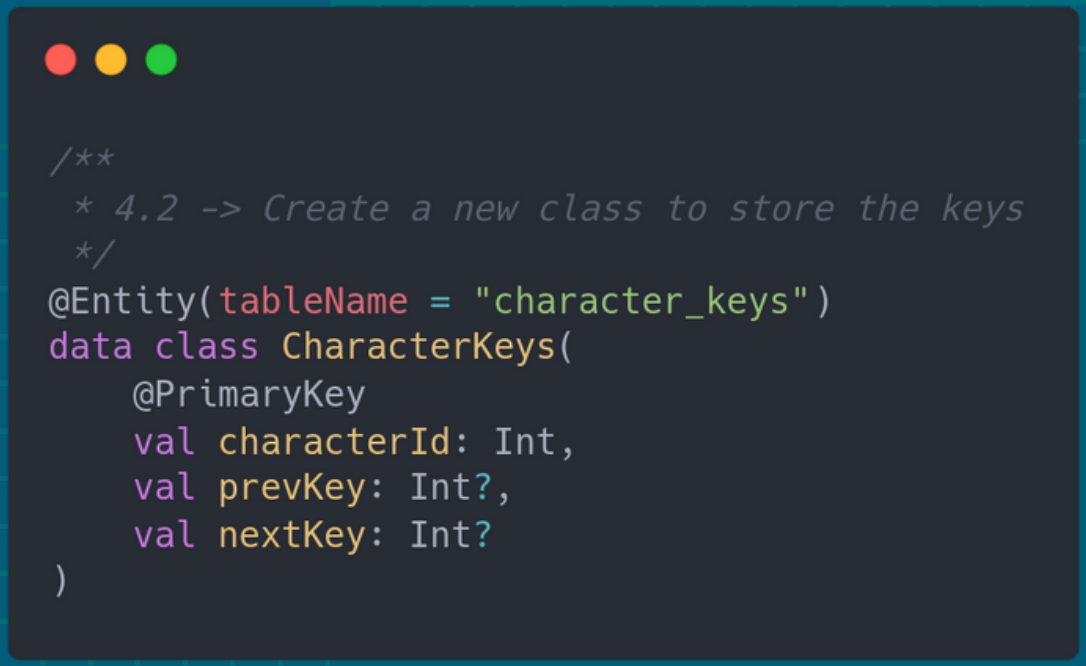
The Database Source of truth

Getting characters from a server and loading them into our database

Convert to Entity

```kotlin
/**
 * 4.1 -> Change to a database entity
 */
@Entity(tableName = "characters")
data class Character(

    @PrimaryKey @SerializedName("id") var id: Int? = null,
    @SerializedName("name") var name: String? = null,
    @SerializedName("status") var status: String? = null,
    @SerializedName("species") var species: String? = null,
    @SerializedName("type") var type: String? = null,
    @SerializedName("gender") var gender: String? = null,
    @SerializedName("image") var image: String? = null,
    @SerializedName("url") var url: String? = null,
    @SerializedName("created") var created: String? = null

)

data class Origin(

    @SerializedName("name") var name: String? = null,
    @SerializedName("url") var url: String? = null

)

data class Location(

    @SerializedName("name") var name: String? = null,
    @SerializedName("url") var url: String? = null

)
```

Entity for Keys



```kotlin
/**
 * 4.2 -> Create a new class to store the keys
 */
@Entity(tableName = "character_keys")
data class CharacterKeys(
    @PrimaryKey
    val characterId: Int,
    val prevKey: Int?,
    val nextKey: Int?
)
```

Remote Mediator

```kotlin
@OptIn(ExperimentalPagingApi::class)
class CharactersRemoteMediator(
    private val service: ApiService,
    private val database: AppDatabase
) : RemoteMediator<Int, Character>() {

    override suspend fun load(
        loadType: LoadType,
        state: PagingState<Int, Character>
    ): MediatorResult {

        val page = when (loadType) {
            LoadType.PREPEND -> {
                val articleKeys = getCharacterKeysForFirstItem(state)
                val prevKey = articleKeys?.prevKey
                if (prevKey == null)
              return MediatorResult.Success(endOfPaginationReached = articleKeys != null)
                prevKey
            }
            LoadType.REFRESH -> {
                val articleKeys = getCharacterKeysClosestToCurrentPosition(state)
                val nextKey = articleKeys?.nextKey
                if (nextKey == null) START_PAGE else nextKey
            }
            LoadType.APPEND -> {
                val articleKeys = getCharacterKeysForLastItem(state)
                val nextKey = articleKeys?.nextKey
                if (nextKey == null)
              return MediatorResult.Success(endOfPaginationReached = articleKeys != null)
                nextKey
            }
        }

        try {

            val response = service.getCharacters(page)
            val characters = response.characters
            val isEndOfPagination = response.info?.next.isNullOrBlank()
//            val isEndOfPagination = articles.isEmpty()

            database.withTransaction {

                if (loadType == LoadType.REFRESH) {
                    database.characterKeysDao().deleteAll()
                    database.charactersDao().deleteAll()
                }

                val prevKey = if (page == START_PAGE) null else page - 1
                val nextKey = if (isEndOfPagination) null else page + 1

                val keys =
                    characters.map { it.id?.let { id -> CharacterKeys(id, prevKey, nextKey) } }

                database.characterKeysDao().insert(keys as List<CharacterKeys>)
                database.charactersDao().insert(characters)

            }

            return MediatorResult.Success(endOfPaginationReached = isEndOfPagination)

        } catch (exception: IOException) {
            return MediatorResult.Error(exception)
        } catch (exception: HttpException) {
            return MediatorResult.Error(exception)
        }
    }
```

First Item Keys

```
    private suspend fun getCharacterKeysForFirstItem(state: PagingState<Int, Character>):
       CharacterKeys? {

        val pagingSource = state.pages.firstOrNull()

        val characters = pagingSource?.data

        if (characters.isNullOrEmpty()) return null

        val firstCharacter = characters.firstOrNull()

        if (firstCharacter == null) return null

        val keys = database.characterKeysDao().getCharacterKeysByCharacterId(firstCharacter.id!!)

        return keys

//        ----------------------------------------------------------------------------------------
//
//        A oneliner for all we've done above. THIS IS KOTLIN!!!
//        ----------------------------------------------------------------------------------------
//
//        return state.pages
//            .lastOrNull() { it.data.isNotEmpty() }?.data?.lastOrNull()
//            ?.let { article ->
//                database.articleKeysDao().getCharacterKeysByCharacterId(article.id)
//            }

    }
```

Last Item Keys

```
    private suspend fun getCharacterKeysForLastItem(state: PagingState<Int, Character>):
      CharacterKeys? {

        val pagingSource = state.pages.lastOrNull()

        val characters = pagingSource?.data

        if (characters.isNullOrEmpty()) return null

        val lastCharacter = characters.lastOrNull()

        if (lastCharacter == null) return null

        val keys = database.characterKeysDao().getCharacterKeysByCharacterId(lastCharacter.id!!)

        return keys

//  ----------------------------------------------------------------------------------------
//        A oneliner for all we've done above. NO WAYYYY!
//  ----------------------------------------------------------------------------------------
//        return state.pages
//            .lastOrNull() { it.data.isNotEmpty() }?.data?.lastOrNull()
//            ?.let { article ->
//                database.articleKeysDao().getCharacterKeysByCharacterId(article.id)
//            }

    }
```

Closest Item Keys

```
private suspend fun getCharacterKeysClosestToCurrentPosition(state: PagingState<Int, Character>):
  CharacterKeys? {

    val anchorPosition = state.anchorPosition

    if (anchorPosition == null) return null

    val closestItem = state.closestItemToPosition(anchorPosition)

    if (closestItem == null) return null

    val keys = database.characterKeysDao().getCharacterKeysByCharacterId(closestItem.id!!)

    return keys

}
```

Repository

```kotlin
/**
 * 2.5 -> Create a character repository passing the ApiService class
 */

class CharactersRepository @Inject constructor(
    private val apiService: ApiService,
    private val database: AppDatabase
) {

    companion object {
        const val PAGE_SIZE = 20
    }

    /**
     * 2.6 -> Create a pager that return the paging data stream
     */
    // REMOTE ONLY
    fun getCharacters(): Flow<PagingData<Character>> {
        return Pager(
            config = PagingConfig(pageSize = PAGE_SIZE, enablePlaceholders = false),
            pagingSourceFactory = { CharactersPagingSource(apiService) }
        ).flow
    }

    /**
     * 4.4 -> Create a pager that returns paging data from database
     */
    // REMOTE WITH CACHE
    @OptIn(ExperimentalPagingApi::class)
    fun getCharactersWithCache(): Flow<PagingData<Character>> {
        return Pager(
            config = PagingConfig(pageSize = PAGE_SIZE, enablePlaceholders = false),
            remoteMediator = CharactersRemoteMediator(service = apiService, database = database),
            pagingSourceFactory = { database.charactersDao().getAllCharacters() }
        ).flow
    }

}
```

12

The Compose Way

Paging the data using declarative UI. Sounds fun, but just wait!

Fragment

```
class ComposeFragment : Fragment() {

    private val viewModel: CharactersViewModel by activityViewModels()

    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View {
        return ComposeView(requireContext()).apply {
            setContent {
                MdcTheme {
                    CharactersScreen(findNavController(), viewModel)
                }
            }
        }
    }

}
```

Container

← Fragment

Fragment

```
@Composable
fun CharactersScreen(
    navController: NavController,
    viewModel: CharactersViewModel
) {

    val characters = viewModel.articlesWithCache.collectAsLazyPagingItems()

    Scaffold(
        topBar = {
            TopAppBar(
                title = { Text(text = "Sweet Compose") },
                navigationIcon = {
                    IconButton(onClick = { navController.popBackStack() }) {
                        Icon(Icons.Filled.ArrowBack, "backIcon")
                    }
                },
                backgroundColor = MaterialTheme.colors.surface,
                contentColor = MaterialTheme.colors.onSurface
            )
        },
        content = {
            CharactersContent(
                modifier = Modifier
                    .padding(it)
                    .fillMaxSize(),
                characters = characters
            )
        }
    )
}
```

Content

Fragment

```
@Composable
fun CharactersContent(modifier: Modifier, characters: LazyPagingItems<Character>) {

    val loadState = characters.loadState

    val errorState = loadState.source.append as? LoadState.Error
        ?: loadState.source.prepend as? LoadState.Error
        ?: loadState.append as? LoadState.Error
        ?: loadState.prepend as? LoadState.Error

    val isEmptyVisible = remember {
        loadState.refresh is LoadState.NotLoading && characters.itemCount == 0
    }
    val isContentVisible = remember {
        loadState.source.refresh is LoadState.NotLoading || loadState.mediator?.refresh is
LoadState.NotLoading
    }
    val isLoadingVisible = remember {
        loadState.mediator?.refresh is LoadState.Loading
    }

    var error = remember {
        errorState?.let { "\uD83D\uDE28 Wooops ${it.error}" }
    }

    Column(modifier = Modifier.fillMaxSize()) {
        CharactersList(modifier = modifier.fillMaxSize(), characters = characters)
    }

}
```

Lazy List

Fragment

```
@Composable
fun CharactersList(modifier: Modifier, characters: LazyPagingItems<Character>) {

    Column {
        Column(
            modifier = Modifier.fillMaxWidth(),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            if (characters.loadState.refresh is LoadState.Loading)
                CircularProgressIndicator(
                    modifier = Modifier
                        .width(16.dp)
                        .height(16.dp)
                        .padding(16.dp, 24.dp)
                )
            if (characters.loadState.refresh is LoadState.Error)
                FloatingActionButton(
                    elevation = FloatingActionButtonDefaults.elevation(0.dp),
                    onClick = { characters.retry() }) {
                    Icon(Icons.Filled.Refresh, "refresh")
                }
        }
        LazyColumn(modifier = Modifier.fillMaxWidth()) {
            items(items = characters, key = { item: Character -> item.id!! }) { value: Character? ->
                value?.let { CharacterItem(character = it) }
            }
        }
        Column(
            modifier = Modifier.fillMaxWidth(),
            verticalArrangement = Arrangement.Center,
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            if (characters.loadState.refresh is LoadState.Loading)
                CircularProgressIndicator(
                    modifier = Modifier
                        .padding(16.dp, 24.dp)
                        .width(16.dp)
                        .height(16.dp)
                )
            if (characters.loadState.refresh is LoadState.Error)
                FloatingActionButton(
                    elevation = FloatingActionButtonDefaults.elevation(0.dp),
                    onClick = { characters.retry() }
                ) {
                    Icon(Icons.Filled.Refresh, "refresh")
                }
        }

    }


}
```

Fragment

```
@OptIn(ExperimentalUnitApi::class)
@Composable
fun CharacterItem(character: Character) {

    val shimmer = Shimmer.AlphaHighlightBuilder()
        .setDuration(1800)
        .setBaseAlpha(0.7f)
        .setHighlightAlpha(0.6f)
        .setDirection(Shimmer.Direction.LEFT_TO_RIGHT)
        .setAutoStart(true)
        .build()

    val shimmerDrawable = ShimmerDrawable().apply {
        setShimmer(shimmer)
    }

    Row(
        modifier = Modifier
            .padding(horizontal = 16.dp, vertical = 8.dp)
            .height(80.dp)
            .fillMaxWidth(),
        verticalAlignment = Alignment.CenterVertically,
        horizontalArrangement = Arrangement.Start
    ) {
        Text(text = "${character.id}")
        Column(
            modifier = Modifier
                .padding(horizontal = 16.dp, vertical = 8.dp)
                .height(72.dp)
                .width(72.dp)
                .clip(CircleShape)
        ) {
            AsyncImage(
                model = ImageRequest.Builder(LocalContext.current)
                    .data(character.image)
                    .crossfade(true)
                    .placeholder(shimmerDrawable)
                    .build(),
                contentDescription = "Character image",
                contentScale = ContentScale.Crop,
            )
        }
        Text(text = "${character.name}")
    }

}
```

Character Item

13

There you go!

What we've learned?

Paging from local, remote, firestore and caching data

14

Any Benefits?

In-memory caching for your paged data. This ensures that your app uses system resources efficiently while working with paged data.

Ensuring that your app uses network bandwidth and system resources efficiently.

Configurable adapters that automatically request data as the user scrolls toward the end of the loaded data.

Built-in support for error handling, including refresh and retry capabilities.

14

Ask Any Question

To learn and Grow is to ask and ask

Question 1

What about storing firestore documents in room so that we can have a local custom cache?

Question 2

Can I add a custom shimmer layout as loading when fetching additional pages?

16

Thank you for joining today's session.

In case for other questions and inquries reach me on twitter @mambo_bryan or through mail : mambobryan@gmail.com