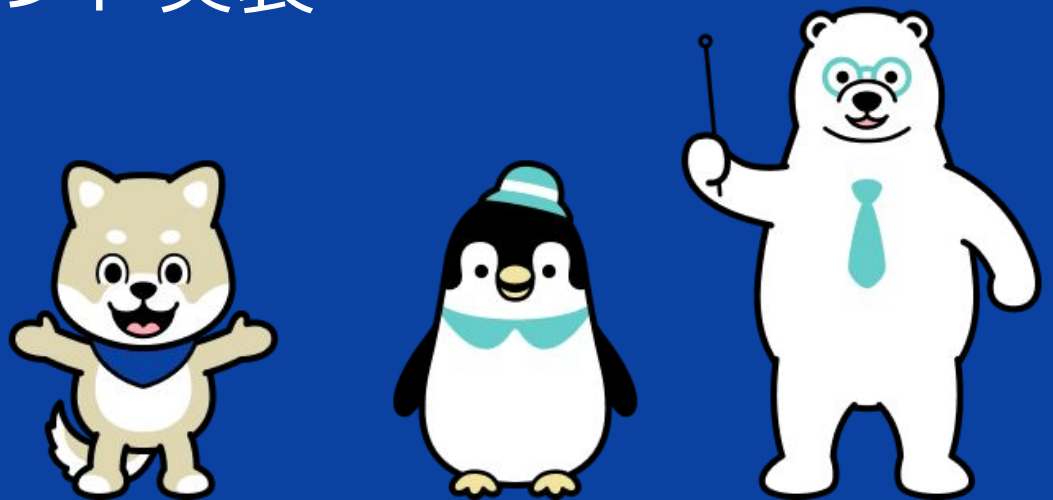
Study Sapuri

DroidKaigi 2024

# GraphQL の魅力を引き出す Android クライアント実装

Kurumi Morimoto (@morux2)



# Kurumi Morimoto (morux2)

StudySapuri at Recruit Co., Ltd.

スタディサプリ小学講座・中学講座の
Android アプリの開発を担当

morux2
_morux2

# スタディサプリ小学講座・中学講座



2024/09/13 時点

# スタディサプリ小学講座・中学講座


- 小学1,2年生 および 中学1~3年生 向けの月額制のオンライン学習サービス
- 2022年2月に中学講座アプリをリニューアル
    - 2023年9月に同アプリに組み込む形で小学講座をリニューアル
- アプリでは学習に辿り着くまでの導線を実装
    - 学習画面は WebView で表示している

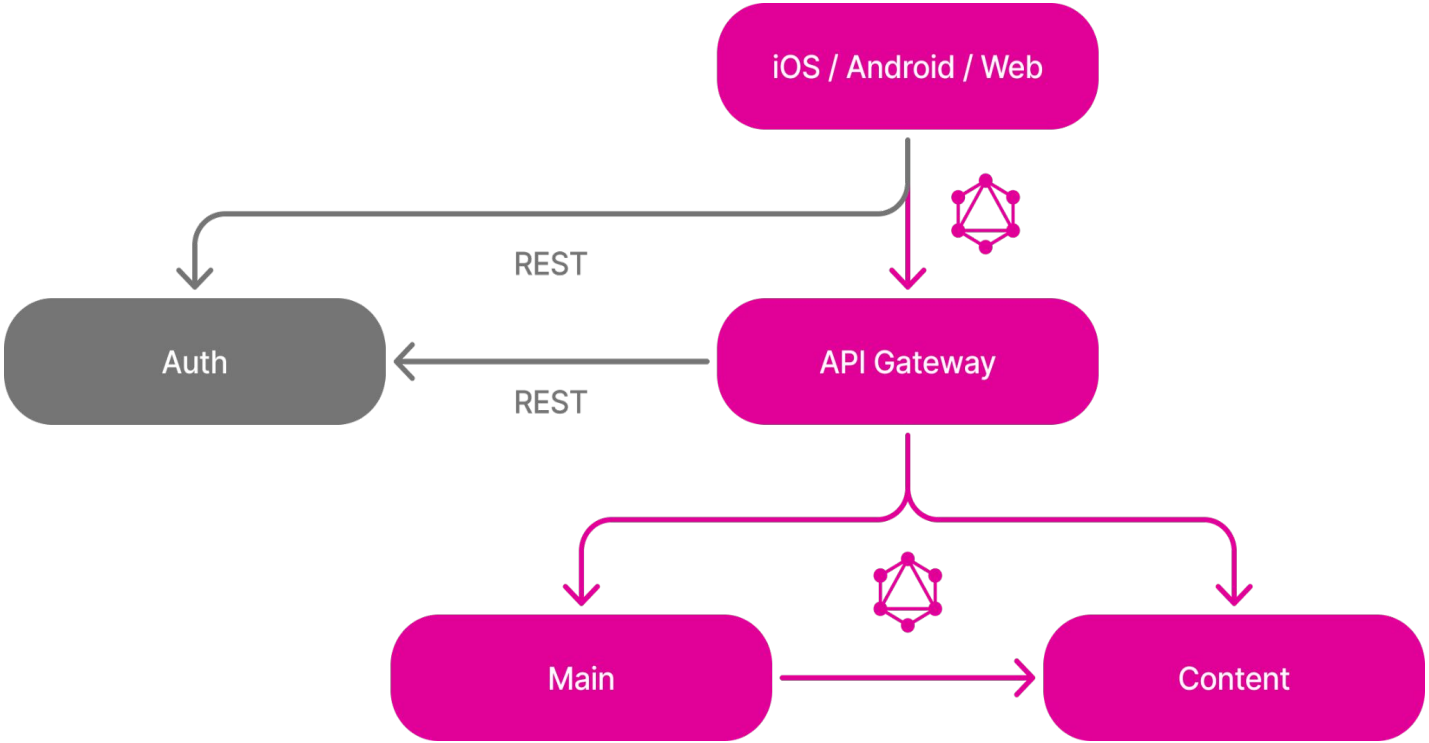
# 00

## スタディサプリ小学講座・中学講座での GraphQL の活用

# スタディサプリ小学講座・中学講座での GraphQL の活用

- スタディサプリ小学・中学講座では GraphQL を全面採用
  - 一部他サービスと共有している箇所(認証など)は, REST API を利用
- クライアントは API Gateway にアクセス
  - API Gateway は GraphQL Schema Stitching で複数のスキーマをマージ
- API Gateway から下層のマイクロサービスにアクセスが振り分けられる
  - ユーザーごとの学習記録や, 学習コンテンツの情報を取得して, クライアントに返却

# スタディサプリ小学講座・中学講座での GraphQL の活用



# GraphQL の採用理由

- クライアントが欲しい情報を柔軟に取得できる
  - オーバーフェッチ / アンダーフェッチの心配がない
  - プラットフォームごとの UI の要求が異なる場合にこの特徴が活きる
- クライアントライブラリの型生成が強力
  - 定義した GraphQL スキーマを元に, クライアントライブラリで型が生成される
  - API スキーマと実装の乖離が起こりにくい
  - => スキーマ駆動開発を加速させる

# 本セッションの目的

GraphQL の導入を検討中の方, 既に GraphQL を導入している方に

スタディサプリ小学講座・中学講座で GraphQL を全面採用し, 2年ほど運用した中で蓄積された ADR (Architecture Decision Records) を紹介し

より効果的に活用するためのノウハウを共有する

## ADR |

01 原則としてドメインロジックはサーバーサイドで実装し，GraphQL Schema 定義に含める

02 Apollo Client の Wrapper クラスを定義し，各画面から呼ぶ GraphQL Query / Mutation を一任する

03 GraphQL Errors と HTTP Status Code のマッピング

04 Fragment Colocation の指針

# 01

## 原則としてドメインロジックはサーバーサイドで実装し，GraphQL Schema 定義に含める

# ドメインロジックは GraphQL Schema 定義に含める

- 教育ドメインでは, クライアント間での挙動の差が大きな問題になり得る
  - リアルタイム性や極端に高いパフォーマンス要求はほとんどない
- サーバーサイドに実装を寄せることで問題を起こりにくくする
- 単にデータソースを返す API にせず, 適切なドメインモデルに落とし込んだ GraphQL Schema を設計する
- 原則通りに実装した場合, クライアント側ではドメインロジックは不要に

# 例)「週ごと」の学習記録を取得する API

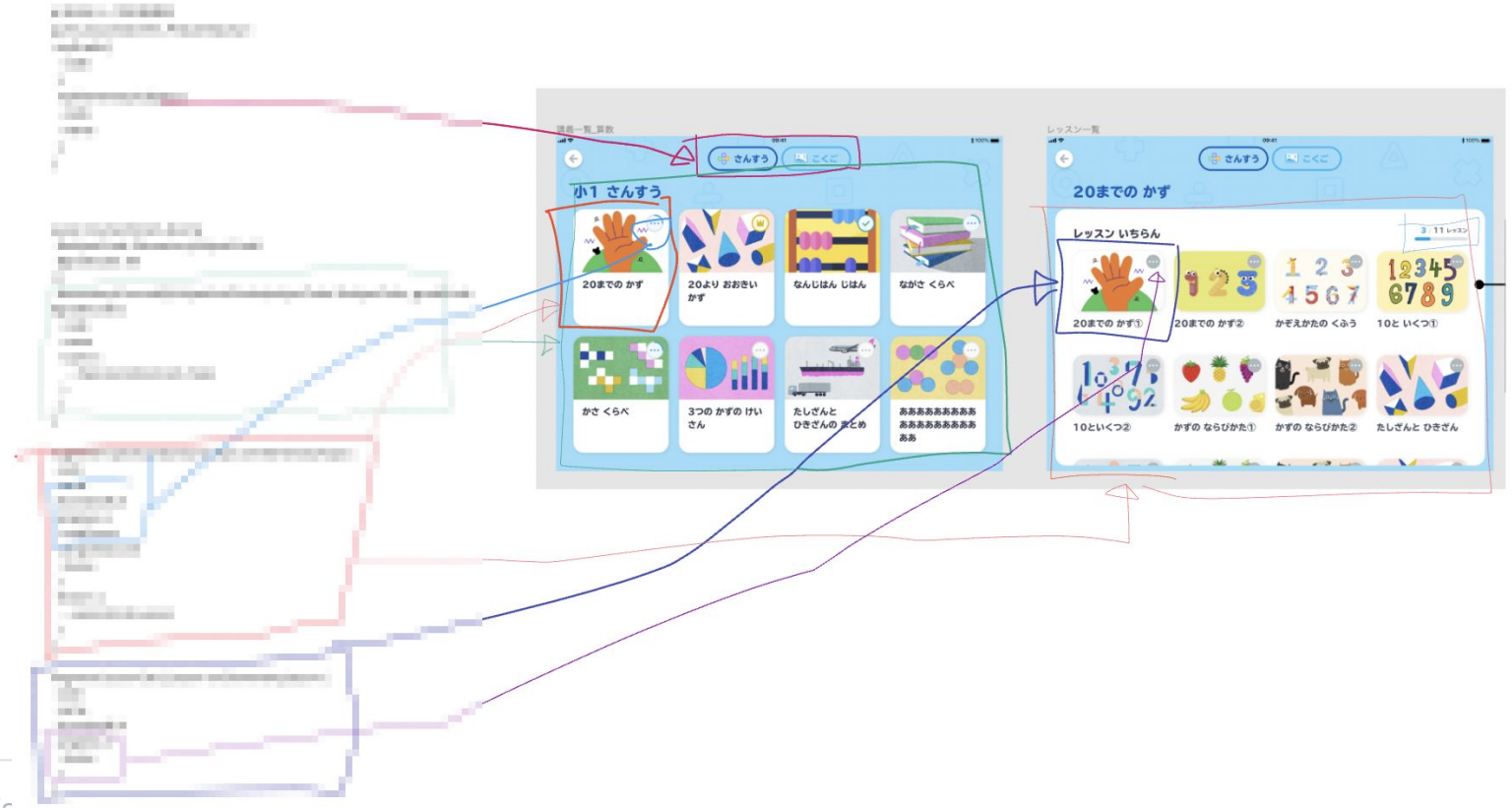
クライアント側で日付の範囲を指定して, 学習記録を取得するような API も設計可能だが,

「集計は月曜開始とする」という仕様をドメインロジックとみなし, サーバーサイド側で計算し, GraphQL Schema に含めた

# (余談) 誰が GraphQL Schema を定義するのか

- スキーマは誰が書いても良い
  - **サーバーサイド専任のタスクではない**
- クライアント / サーバーサイド のメンバーでレビューし合って定義
  - よりクライアント主体のデータフェッチを実現でき, GraphQL の特性を活かしやすい

# (余談) クライアント / サーバーサイド の協働の様子



# (余談) クライアント / サーバーサイド の協働の様子

# クライアントでは, 原則ドメインロジックの計算をしない

- 推奨アーキテクチャでは，UI レイヤとデータレイヤの2つが存在
  - UI レイヤ：画面にアプリデータを表示
  - データレイヤ：ビジネスロジックを含み, アプリデータを公開

https://developer.android.com/topic/architecture?hl=ja#modern-app-architecture

# クライアントでは, 原則ドメインロジックの計算をしない

- 推奨アーキテクチャでは，UI レイヤとデータレイヤの2つが存在
  - UI レイヤ：画面にアプリデータを表示
  - ~~データレイヤ：ビジネス ロジックを含み, アプリデータを公開~~

**GraphQL Schema がこの役割を担うため,**
**クライアントは実装不要**

# クライアントは UI レイヤを実装する

- UI レイヤは UI 要素と状態ホルダで構成
  - UI 要素： Jetpack Compose
  - 状態ホルダ：AAC の ViewModel (ViewModel の廃止検討については付録で紹介)

https://developer.android.com/topic/architecture?hl=ja#ui-layer

# クライアントは UI レイヤを実装する

無理に推奨アーキテクチャに当てはめない

- Apollo Client をラップした, 実態にそぐわない Repository や UseCase を定義しない
- GraphQL で完結せず, ローカル / リモート Data Source にアクセスする場合に, 適切に Repository や UseCase を用いる
    - ローカル Data Source : Room, Preference DataStore
    - リモート Data Source : REST API with Retrofit

# ここまでのまとめ

- 原則としてドメインロジックはサーバーサイドで実装し, GraphQL Schema 定義に含める
- GraphQL で完結する画面については, 推奨アーキテクチャに無理に当てはめずに, UI レイヤのみを実装する
- ローカル / リモート DataSource にアクセスする箇所は, 推奨アーキテクチャを参考に, 適切にドメイン / データレイヤを実装する

# サンプルコード


https://github.com/quipper/droidkaigi-2024-graphql-android-client-sample

# サンプルコード (ライブラリ)

```
[versions]
agp = "8.5.2"
kotlin = "1.9.0"
coreKtx = "1.13.1"
lifecycleRuntimeKtx = "2.8.4"
activityCompose = "1.9.1"
composeBom = "2024.08.00"
kotlinxCoroutines = "1.8.1"
timber = "5.0.1"
apollo = "3.8.4"
hilt = "2.51.1"
androidxHiltNavigationCompose = "1.2.0"
junitBom = "5.10.3"
kotest = "5.9.1"
mockk = "1.13.12"
```

GraphQL クエリから Kotlin のモデルを生成する
Kotlin 向けの GraphQL クライアントライブラリ

# サンプルコード (Query)

```
query MainScreenQuery {
    posts {
        ...PostCardFragment
    }
}

fragment PostCardFragment on Post {
    author {
        name
    }
    title
    body
}
```

# サンプルコード (Apollo に自動生成された型)

```
public class MainScreenQuery() : Query<MainScreenQuery.Data> {
  public override fun equals(other: Any?): Boolean = other != null && other::class == this::class

  public override fun hashCode(): Int = this::class.hashCode()

  public override fun id(): String = OPERATION_ID

  public override fun document(): String = OPERATION_DOCUMENT

  public override fun name(): String = OPERATION_NAME
```

# サンプルコード (Apollo に自動生成された型)

```
@ApolloAdaptableWith(MainScreenQuery_ResponseAdapter.Data::class)
public data class Data(
  public val posts: List<Post>,
) : Query.Data

public data class Post(
  public val __typename: String,
  /**
   * Synthetic field for 'PostCardFragment'
   */
  public val postCardFragment: PostCardFragment,
)
```

# サンプルコード (Apollo Client)

```
object ApolloClientFactory {  morux2

    private const val BASE_URL = "http://10.0.2.2:4000/"

    fun from(context: Context): ApolloClient {  morux2
        return ApolloClient.Builder()
            .okHttpClient(
                OkHttpClient().newBuilder().authenticator(MyAuthenticator()).build()
            )
            .serverUrl(BASE_URL)
            .addInterceptor(ApolloNetworkConnectivityInterceptor(context))
            .build()
    }
}
```

# サンプルコード (ApolloWrapper)

```
class MainApolloWrapper @Inject constructor(  new *
    private val apolloClient: ApolloClient
) {
    fun fetchMainScreenData(): Flow<MainScreenQuery.Data> {  new
        return apolloClient.query(MainScreenQuery())
            .toThrowableFlow()
    }
}
```

# サンプルコード (ViewModel)

```
@HiltViewModel  new *
class MainViewModel @Inject constructor(
    private val apolloWrapper: MainApolloWrapper
) : ViewModel() {
    private val _viewState = MutableStateFlow(ViewState.INITIAL)
    val viewState = _viewState.asStateFlow()

    fun fetchContent() {  new *
        apolloWrapper.fetchMainScreenData().toLce().onEach { lce ->
            _viewState.update {
                it.copy(
                    isLoading = lce.isLoading,
                    throwable = lce.getThrowableIfError(),
                    content = lce.getDataIfContent()
                )
            }
        }.launchIn(viewModelScope)
    }
```

# サンプルコード (View)

```
@Composable  morux2
fun MainScreen(
    viewModel: MainViewModel = hiltViewModel()
) {
    val viewState by viewModel.viewState.collectAsStateWithLifecycle()

    LifecycleEventEffect(Lifecycle.Event.ON_RESUME) {
        viewModel.fetchContent()
    }

    MainScreenContent(
        modifier = Modifier.fillMaxSize(),
        viewState = viewState,
        refetch = viewModel::fetchContent,
    )
}
```

# 02

## Apollo Client の Wrapper クラスを定義し，
## 各画面から呼ぶ GraphQL Query / Mutation を一任する

# Apollo Client の Wrapper クラスを定義する

- Apollo Client をラップした ApolloWrapper クラスを各画面に定義し, 各画面から呼ぶ GraphQL Query / Mutation を一任する
- ApolloWrapper は Flow<T> で 値を返却し, エラーの場合は例外を投げる
- ViewModel は 直接 Apollo Client を叩かず, ApolloWrapper を呼び出す
- ViewModel と同じ階層に配置し, UI レイヤであることを強調する
- ApolloWrapper は Apollo Client 以外を引数にとらない

# Apollo Client の Wrapper クラスを定義する

ApolloWrapper は Apollo Client 以外を引数にとらない

```
class MainApolloWrapper @Inject constructor(  new *
    private val apolloClient: ApolloClient
) {
    fun fetchMainScreenData(): Flow<MainScreenQuery.Data> {
        return apolloClient.query(MainScreenQuery())
            .toThrowableFlow()
    }
}
```

# Apollo Client の Wrapper クラスを定義する

Apollo Client は Flow<T> で 値を返却し, エラーの場合は例外を投げる

```
class MainApolloWrapper @Inject constructor(  new *
    private val apolloClient: ApolloClient
) {
    fun fetchMainScreenData(): Flow<MainScreenQuery.Data> {
        return apolloClient.query(MainScreenQuery())
            .toThrowableFlow()
    }
}
```

# Apollo Client の Wrapper クラスを定義する

Apollo Client は Flow<T> で 値を返却し, エラーの場合は例外を投げる

```
class MainApolloWrapper @Inject constructor(  new *
    private val apolloClient: ApolloClient
) {
    fun fetchMainScreenData(): Flow<MainScreenQuery.Data> {
        return apolloClient.query(MainScreenQuery())
            .toThrowableFlow()
    }
}
```

# Apollo のレスポンスをどうやって Flow<T> に変換するか

## Apollo の toFlow() 関数 を呼び出す

common

```
fun toFlow(): Flow<ApolloResponse<D>>
```

Returns a cold Flow that produces ApolloResponses for this ApolloCall. Note that the execution happens when collecting the Flow. This method can be called several times to execute a call again.

Example:

```
apolloClient.subscription(NewOrders())
                .toFlow()
                .collect {
                  println("order received: ${it.data?.order?.id"})
                }
```

https://www.apollographql.com/docs/kotlin/kdoc/older/3.8.2/apollo-runtime/com.apollographql.apollo3/-apollo-call/to-flow.html

# Apollo のレスポンスをどうやって Flow<T> に変換するか

ApolloResponse には, data と errors が格納されている

errors が存在する場合は例外を投げつつ, Flow で data を返却する (後述)

```
@JvmField
val data: D?
```

Parsed response of GraphQL operation execution. Can be `null` in case if operation execution failed.

```
@JvmField
val errors: List<Error>?
```

GraphQL operation execution errors returned by the server to let client know that something has gone wrong. This can either be null or empty depending on what your server sends back

https://www.apollographql.com/docs/kotlin/kdoc/older/3.8.2/apollo-api/com.apollographql.apollo3.api/-apollo-response/index.html

# Apollo Client の Wrapper クラスを定義する

ApolloWrapper をクラスを定義している理由は2つ

- 画面の描画に複数の Query を叩く必要がある場合に, ApolloWrapper が値のマージ処理を担うことで, ViewModel の処理をシンプルに保つ
    - **本来はマージが必要になる Schema にすべきではないが, サーバーサイドの都合でやむを得ない場合がある. Query の結果をそのまま画面に流すのが理想.**
- ViewModel のテストの容易性が上がる
    - Apollo Client よりも ApolloWrapper の方が mock しやすい

# ViewModel の処理をシンプルに保つ

```
fun fetchMainScreenData(): Flow<List<Pair<PostQuery.Post, AuthorQuery.Author>>> {
    return fetchPosts().flatMapLatest { posts ->
        val postFlows = posts.map { post ->
            // post に対応する author の情報を取得する
            fetchAuthor(authorId = post.author.id).map { author ->
                author?.let {
                    Pair(post, it)
                }
            }
        }
        combine(postFlows) {
            it.toList().filterNotNull()
        }
    }
}
```

Flow の平坦化や加工は ApolloWrapper で実施し
ViewModel の見通しを良くする

# ViewModel のテストの容易性が上がる

```
@Test  new *
fun 画面が描画できること() = runTest {
    val viewModel = viewModelFactory(
        apolloClient = mockk {
            every { query(MainScreenQuery()).toFlow() } returns flowOf(
                ApolloResponse.Builder(
                    requestUuid = mockk(),
                    operation = mockk<MainScreenQuery>(),
                    data = dummyContent
                ).errors(null).build()
            )
        }
    )

    viewModel.fetchContent()
    advanceUntilIdle()
```

Apollo Client の mock を作成する場合は
ダミーのレスポンスを返却する

mock 処理が肥大化しやすい

# ViewModel のテストの容易性が上がる

```
@Test  new *
fun エラーの場合はダイアログを表示すること() = runTest {
    val viewModel = viewModelFactory(
        apolloClient = mockk {
            every { query(MainScreenQuery()).toFlow() } returns flowOf(
                ApolloResponse.Builder(
                    requestUuid = mockk(),
                    operation = mockk<MainScreenQuery>(),
                    data = null,
                ).errors(listOf(mockk(relaxed = true))).build()
            )
        }
    )

    viewModel.fetchContent()
    advanceUntilIdle()
```

Apollo Client の mock を作成する場合は
ダミーのレスポンスを返却する

mock 処理が肥大化しやすい

# ViewModel のテストの容易性が上がる

```
@Test  new *
fun 画面が描画できること() {
    val viewModel = viewModelFactory(
        apolloWrapper = mockk {
            every { fetchMainScreenData() } returns flowOf(dummyContent)
        }
    )

    viewModel.fetchContent()

    viewModel.viewState.value shouldBe MainViewModel.ViewState(
        isLoading = false,
        throwable = null,
        content = dummyContent,
    )
}
```

ApolloWrapper を mock すると
処理がシンプルで流れを辿りやすい

# ViewModel のテストの容易性が上がる

```
@Test  morux2 *
fun エラーの場合はダイアログを表示すること() {
    val exception = GraphQlServerException(listOf(mockk(relaxed = true)))
    val viewModel = viewModelFactory(
        apolloWrapper = mockk {
            every { fetchMainScreenData() } returns flow { throw exception }
        }
    )

    viewModel.fetchContent()

    viewModel.viewState.value shouldBe MainViewModel.ViewState(
        isLoading = false,
        throwable = exception,
        content = null,
    )
}
```

ApolloWrapper を mock すると
処理がシンプルで流れを辿りやすい

# ViewModel でレスポンスを束ね, 画面の状態を計算する

- ViewModel は, ApolloWrapper / Repository / UseCase から 受け取った Flow を束ね, Loading / Content / Error の状態を計算する
    - ApolloWrapper : GraphQL with Apollo
    - Repository : Room, Preference DataStore, REST API with Retrofit
    - UseCase : 複数画面から呼び出されるドメインロジック (ログイン / ログアウト処理など)
- 状態は LCE の Sealed Class で表現し, StateFlow で流す

# 状態は LCE の Sealed Class で表現する

```
fun <T> Flow<T>.toLce() = map<T, Lce<T>> {  new *
    Lce.Content(it)
}.onStart {
    emit(Lce.Loading)
}.catch {
    emit(Lce.Error(it))
}
```

# 状態は LCE の Sealed Class で表現する

```
sealed class Lce<out T> {  new *
    data object Loading : Lce<Nothing>()  new *
    data class Content<T>(val data: T) : Lce<T>()  new *
    data class Error(val throwable: Throwable) : Lce<Nothing>()  ne

    val isLoading  new *
        get() = this is Loading

    fun getDataIfContent() = (this as? Content)?.data  new *
    fun getThrowableIfError() = (this as? Error)?.throwable  new *
}
```

# ViewModel で LCE を計算する

```
@HiltViewModel  new *
class MainViewModel @Inject constructor(
    private val apolloWrapper: MainApolloWrapper
) : ViewModel() {
    private val _viewState = MutableStateFlow(ViewState.INITIAL)
    val viewState = _viewState.asStateFlow()

    fun fetchContent() {  new *
        apolloWrapper.fetchMainScreenData().toLce() onEach { lce ->
            _viewState.update {
                it.copy(
                    isLoading = lce.isLoading,
                    throwable = lce.getThrowableIfError(),
                    content = lce.getDataIfContent()
                )
            }
        }.launchIn(viewModelScope)
    }
```

# 例) ViewModel で複数の Flow を束ねる

```
@HiltViewModel
class MyPageProfileViewModel @Inject constructor(
    private val apolloWrapper: MyPageProfileApolloWrapper,
    private val getBillingInfo: GetBillingInfoUseCase,
) : ViewModel() {

    private val _viewState = MutableStateFlow(ViewState.INITIAL)
    val viewState = _viewState.asStateFlow()

    fun requestMyPageProfileData() {  morux2 +2 *
        combine(
            apolloWrapper.getMyPageProfileData(),
            getBillingInfo(),
            ::Pair
        ).toLce().map { data ->
            _viewState.update {
                it.copy(
                    isLoading = data.isLoading,
                    error = data.getThrowableIfError()?.let { error -> Error.GetMyPageProfileDataError(error) },
                    myPageProfileQueryData = data.getDataIfContent()?.first,
                    billingInfo = data.getDataIfContent()?.second,
                )
            }
        }.launchIn(viewModelScope)
    }
```

# 例) ViewModel で複数の Flow を束ねる

```
@Test
fun マイページが描画できること() {
    val myPageProfileData = mockk<MyPageProfileQuery.Data>()
    val billingInfo = BillingInfo.IabSubscribed( billingUrl: "dummy url")

    val viewModel = viewModelFactory(
        apolloWrapper = apolloWrapperFactory(myPageProfileQueryFlow = flowOf(myPageProfileData)),
        getBillingInfoUseCase = getBillingInfoUseCaseFactory(billingInfoFlow = flowOf(billingInfo)),
    )

    viewModel.requestMyPageProfileData()

    viewModel.viewState.value shouldBe MyPageProfileViewModel.ViewState(
        isLoading = false,
        error = null,
        myPageProfileQueryData = myPageProfileData,
        billingInfo = billingInfo,
    )
}
```

# ここまでのまとめ

- 画面ごとに ApolloWrapper クラスを定義し, GraphQL の処理を一任する
- ApolloWrapper は, ApolloResponse の data を Flow で返却し, errors が存在する場合は例外を投げる
- ViewModel では, ApolloWrapper / UseCase / Repository の結果を束ね, LCE (Loading / Content / Error) の状態を計算する

# 03

## GraphQL Errors と HTTP Status Code のマッピング

# GraphQL Errors と HTTP Status Code のマッピング

- 公式の GraphQL Over HTTP では, Errors の有無と HTTP Status に関係を持たせず, 常に 200 を返すように推奨されている
  - 部分的なエラーの場合に, 処理を継続できる可能性がある
  - クライアント側が HTTP レイヤを気にする必要がないように

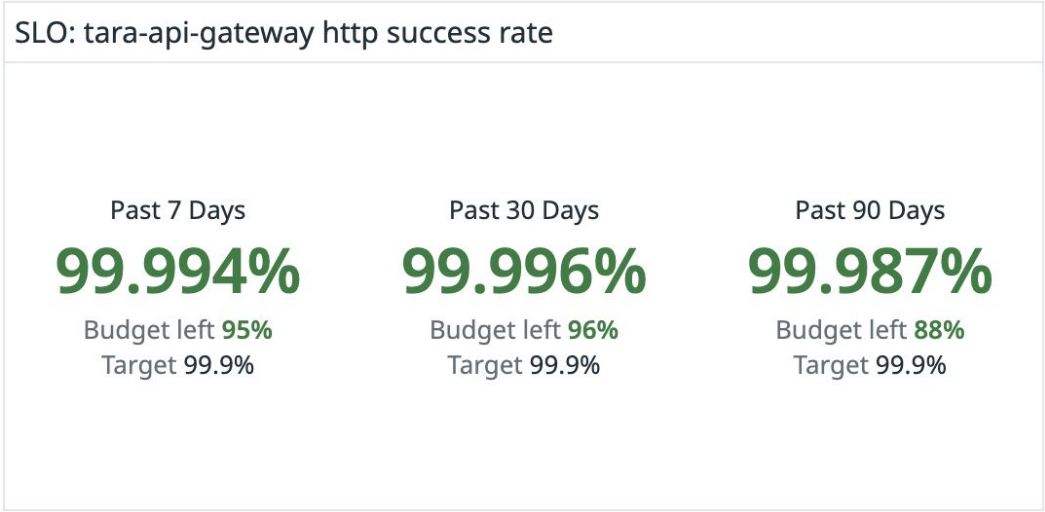
https://graphql.github.io/graphql-over-http/draft/#sec-application-json

# GraphQL Errors と HTTP Status Code のマッピング

- しかし, 我々はクライアント起因のエラーの場合は, Status Code 400, サーバーサイド起因の場合は 500 を返し, GraphQL Errors にエラーを含めるという運用を選択している
    - サービスの構成上, 部分エラーの場合に処理を継続することが困難
    - 一部 REST API を呼び出している箇所があり, クライアントが HTTP レイヤを気にする必要はある
    - 監視と相性がいい

# (余談) API Gateway の Http Success Rate

7 / 30 / 90 日間で, 全リクエストのうちの何%が 200 リクエストで返ってきているかを, SLO として監視している



# クライアントに起因するエラーの場合

- HTTP リクエストを送信する前後でエラーをハンドリングする
  - ネットワーク接続がない場合 (リクエスト前)
  - 認証エラー (401) の場合 (リクエスト後)

# ネットワーク接続がない場合 (リクエスト前)

ネットワーク接続がない場合は, リクエスト前に Apollo で Intercept する

```
object ApolloClientFactory {  morux2

    private const val BASE_URL = "http://10.0.2.2:4000/"

    fun from(context: Context): ApolloClient {  morux2
        return ApolloClient.Builder()
            .okHttpClient(
                OkHttpClient().newBuilder().authenticator(MyAuthenticator()).build()
            )
            .serverUrl(BASE_URL)
            .addInterceptor(ApolloNetworkConnectivityInterceptor(context))
            .build()
    }
}
```

# ネットワーク接続がない場合 (リクエスト前)

```
class ApolloNetworkConnectivityInterceptor(private val context: Context) : ApolloInterceptor {
    override fun <D : Operation.Data> intercept(  new *
        request: ApolloRequest<D>,
        chain: ApolloInterceptorChain
    ): Flow<ApolloResponse<D>> {
        return if (context.isNetworkConnected()) {
            chain.proceed(request)
        } else {
            flow { throw NoNetworkException() }
        }
    }

    private fun Context.isNetworkConnected(): Boolean =  new *
        getSystemService(ConnectivityManager::class.java)?.activeNetwork != null
}
```

# 認証エラー (401) の場合 (リクエスト後)

認証エラーの場合は, OkHttpClient の Authenticator でハンドリングする

```
object ApolloClientFactory {  morux2

    private const val BASE_URL = "http://10.0.2.2:4000/"

    fun from(context: Context): ApolloClient {  morux2
        return ApolloClient.Builder()
            .okHttpClient(
                OkHttpClient().newBuilder().authenticator(MyAuthenticator()).build()
            )
            .serverUrl(BASE_URL)
            .addInterceptor(ApolloNetworkConnectivityInterceptor(context))
            .build()
    }
}
```

# 認証エラー (401) の場合 (リクエスト後)

OkHttpClient の Authenticator を用いると, 401 エラーが返却された場合に, 任意の処理を実行できる.

認証情報をヘッダに付与したリクエストを返却すると自動でリトライが実行され, null を返却するとリトライをスキップする

```
// 401 Unauthorized が返ってきたときに呼び出される
class MyAuthenticator : Authenticator {  morux2
    override fun authenticate(route: Route?, response: Response): Request? {  morux2
        // 本来はここで，ヘッダに認証情報を追加してリクエストをやり直したり，認証情報を付与できない場合はログアウト等のコールバック処理を呼び出す
        Timber.d( message: "response: $response")
        return null
    }
}
```

https://square.github.io/okhttp/recipes/#handling-authentication-kt-java

# サーバーサイドに起因するエラーの場合

toThrowableFlow() という関数を独自で定義し, Errors が存在する場合は, 例外を throw する (Errors の中身を見たロジックは組んでいない)

```
fun <T : Operation.Data> ApolloCall<T>.toThrowableFlow() = toFlow().map { response ->
    response.errors?.let {
        throw GraphQlServerException(it)
    } ?: run {
        response.data ?: run {
            val exception = GraphQlNoDataException()
            throw exception
        }
    }
```

# サーバーサイドに起因するエラーの場合

toThrowableFlow() にはリトライの機構を導入している

Exponential Backoff でリトライ間隔を徐々に伸ばしている

```
}.retryWhen { cause, attempt ->
    if (cause is NoNetworkException || attempt >= 5) return@retryWhen false
    // exponential backoff でリトライ間隔を徐々に伸ばす
    val delayMillis = (300 * 2.0.pow(attempt.toDouble()) * Math.random()).toLong()
    delay(delayMillis)
    Timber.d( message: "retryCount: ${attempt + 1}, delayMills: $delayMillis")
    true
}
```

# ここまでのまとめ

- クライアント起因のエラーの場合は, HTTP Status Code 400, サーバーサイド起因の場合は 500 を返却し, GraphQL Errors にエラーを含める
- クライアント起因のエラーは, リクエスト前後で適切にハンドリングする
- サーバーサイド起因のエラーの場合は, リトライ処理をしつつ, エラーの中身は確認せずに, 例外を投げる

# 04

## Fragment Colocation の指針

# GraphQL Fragment

GraphQL Query / Mutation では, Fragment という再利用可能なフィールドの集合を定義できる

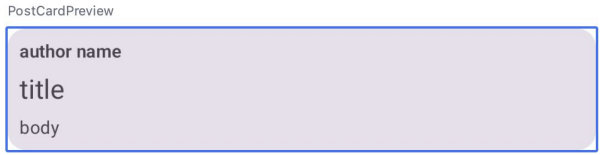
```
fragment PostCardFragment on Post {
    author {
        name
    }
    title
    body
}
```

```
public data class PostCardFragment(
  public val author: Author,
  public val title: String,
  public val body: String,
) : Fragment.Data {
  public data class Author(
    public val name: String,
  )
}
```

# GraphQL Fragment Colocation

GraphQL Fragment と UI コンポーネントを「一緒に配置する」(同一ファイルに定義する) ことを, Fragment Colocationと呼ぶ

```
fragment PostCardFragment on Post {
    author {
        name
    }
    title
    body
}
```




```
@Composable  new *
fun PostCard(
    postCardFragment: PostCardFragment,
    modifier: Modifier = Modifier,
) {
    Card(modifier = modifier) {
        Column(
            modifier = Modifier.padding(all = 8.dp),
            verticalArrangement = Arrangement.spacedBy(8.dp)
        ) {
```

# Fragment Colocation のメリット

- UI コンポーネントが必要とするデータがわかりやすい
- あるコンポーネントが必要とするデータが増減した時に, 修正箇所がそのコンポーネント内に閉じるため, メンテナンス性が向上する
- フィールドの増減に気づきやすいので, オーバーフェッチを防げる

# Apollo Kotlin Plugin

Apollo Kotlin を利用している以上, GraphQL Fragment の定義と UI コンポーネントを同一ファイルに記載することはできない

Apollo Kotlin plugin を使うことで, 定義元の GraphQL ファイルに簡単にジャンプできるので，同一ファイルに記載することにこだわる必要はない

https://www.apollographql.com/blog/announcing-the-apollo-kotlin-plugin-for-android-studio-and-intellij-idea

# Apollo Kotlin Plugin

```
@Composable  morux2
fun PostCard(
    postCardFragment: PostCardFragment,
    modifier: Modifier = Modifier,
) {
    Card(modifier = modifier) {
        Column(
            modifier = Modifier.padding(all = 8.dp),
            verticalArrangement = Arrangement.spacedBy(8.dp)
        ) {
            Text(
                text = postCardFragment.author.name,
                fontSize = 12.sp,
                fontWeight = FontWeight.SemiBold
            )
            Text(
                text = postCardFragment.title,
                fontSize = 18.sp
            )
            Text(
                text = postCardFragment.body,
                fontSize = 12.sp
            )
```

```
18  fun PostCard(
19      postCardFragment: PostCardFragment,
20      modifier: Modifier = Modifier,
```

PostCardFragment GraphQL definition

# Fragment Colocation の指針 ①

- Composable 関数が GraphQL Query / Mutation の フィールドに依存している場合は, Preview の単位にあわせて Fragment を切ること
    - Preview をする場合は, フィールドが1つの場合でも Fragment として切り出す
    - Preview よりも細かい単位で Fragment を切り出しても良いが, Fragment を切り出しすぎると, 階層が深くなり逆にコードの見通しが悪くなる場合があるので注意する
- Composable 関数 と Fragment の対応関係を明確にするため, Fragment の命名は, Composable 関数名 + Fragment とする

# Fragment Colocation

```
@Composable  new *
fun PostCard(
    postCardFragment: PostCardFragment,
    modifier: Modifier = Modifier,
) {
    Card(modifier = modifier) {
        Column(
            modifier = Modifier.padding(all = 8.dp),
            verticalArrangement = Arrangement.spacedBy(8.dp)
        ) {
            Text(
                text = postCardFragment.author.name,
                fontSize = 12.sp,
                fontWeight = FontWeight.SemiBold
            )
            Text(
                text = postCardFragment.title,
                fontSize = 18.sp
            )
            Text(
                text = postCardFragment.body,
                fontSize = 12.sp
            )
```

PostCardPreview


```
fragment PostCardFragment on Post {
    author {
        name
    }
    title
    body
}
```

# Fragment Colocation の指針 ②

- GraphQL Query / Mutation の上の構造が同じであっても, UI コンポーネントが異なる場合は, 別々の Fragment を定義する
- UI コンポーネントと関係のない Fragment を共有しない

# Fragment Colocation

```kotlin
@Composable
fun CourseCard(
    fragment: CourseCardFragment,
    onCourseClicked: (courseCode: String) -> Unit,
    modifier: Modifier = Modifier,
) {
    Card(
        modifier = modifier.fillMaxWidth(),
        shape = Radius.SIZE_6.shape,
        colors = CardDefaults.cardColors(
            containerColor = JuniorHighColor.semantic.surface.main,
            contentColor = JuniorHighColor.semantic.text.main,
        ),
        elevation = CardDefaults.cardElevation(
            defaultElevation = Shadow.SIZE_1.e,
        ),
        onClick = { onCourseClicked(fragment.code) }
    ) {
        Row(
            modifier = Modifier
                .height(IntrinsicSize.Min)
                .fillMaxWidth()
                .padding(Spacing.SIZE_16.size)
        ) {
            TitleArea(
                modifier = Modifier
                    .defaultMinSize(minHeight = 60.dp)
                    .fillMaxHeight()
                    .weight(1f),
                fragment = fragment.courseCardTitleAreaFragment
            )
            fragment.courseMissionInfo?.courseCardMissionAreaFragment?.let {
                MissionArea(fragment = it)
            }
        }
    }
```

# Fragment Colocation

```
@Composable
private fun CourseCardTitleArea(
    fragment: CourseCardTitleAreaFragment,
    modifier: Modifier = Modifier,
) {
    Column(modifier = modifier) {
        Box(
            modifier = Modifier
                .fillMaxHeight()
                .weight(1f),
            contentAlignment = Alignment.Center
        ) {
            Text(
                text = fragment.shortName,
                style = Typography.Default.Bold.size18.copy(color = JuniorHighColor.semantic.text.main)
            )
        }
        Text(
            text = fragment.subtitle.orEmpty(),
            style = Typography.Default.Bold.size12.copy(color = JuniorHighColor.semantic.text.main)
        )
    }
```

# Fragment Colocation

```
@Composable
private fun CourseCardMissionArea(
    fragment: CourseCardMissionAreaFragment,
    modifier: Modifier = Modifier,
) {
    if (fragment.allCompleted) {
        Column(
            modifier = modifier.padding(start = Spacing.SIZE_16.size),
            horizontalAlignment = Alignment.CenterHorizontally
        ) {
            Image(
                modifier = Modifier.height(52.dp),
                painter = painterResource(id = R.drawable.ic_mission_all_completed),
                contentDescription = null
            )
            Text(
                modifier = Modifier.padding(top = Spacing.SIZE_4.size),
                text = stringResource(id = "ミッション完了！"),
                style = Typography.Default.Bold.size12.copy(color = JuniorHighColor.semantic.success._900)
            )
        }
    } else if (fragment.shouldShowMissionInfo) {
        Column(
            modifier = Modifier
                .width(IntrinsicSize.Min)
                .fillMaxHeight()
                .padding(start = Spacing.SIZE_16.size),
            horizontalAlignment = Alignment.CenterHorizontally,
            verticalArrangement = Arrangement.Center
        ) {
            Row {
                Text(
                    text = stringResource(id = "残りミッション数"),
                    style = Typography.Default.Bold.size12.copy(color = JuniorHighColor.semantic.text._600),
                    modifier = Modifier.alignByBaseline()
                )
                Text(
                    text = fragment.remainingMissionCountString,
                    modifier = Modifier
                        .padding(start = Spacing.SIZE_8.size)
```

# ここまでのまとめ

- Fragment Colocation を実践すると, コードのメンテナンス性やパフォーマンスの向上が期待できる
- Apollo Kotlin Plugin を使うと, 定義元の GraphQL ファイルに簡単にジャンプできる
- Composable 関数の Preview の単位で Fragment Colocation をする
- UI コンポーネントと関係のない単位で Fragment を切るのは, オーバーフェッチの原因になるので避ける

# 総括

GraphQL の魅力を引き出すには....

- ドメインモデルを反映させた GraphQL Schema を設計し
- 適切に UI レイヤから Apollo Client を呼び出し
- プロダクトの特性を考慮してエラーハンドリングを設計し
- Composable 関数と GraphQL Fragment を対応させる

ことが大切です！

# Thank you for listening !!


**Kurumi Morimoto (morux2)**

StudySapuri at Recruit Co., Ltd.

morux2
_morux2


付録には以下2点を掲載しています

- Persisted Query に変わるクエリ信頼の仕組みとして Signed Query を採用する
- ViewModel 廃止検討

# Appendix

# 05

## Persisted Query に変わるクエリ信頼の仕組みとして Signed Query を採用する

# クエリ信頼の仕組みとして Signed Query を採用する

2024-05-27

## GraphQL Trusted Documents の実装パターンを Signed Query に移行しました (Android編)

Engineering Android Native

こんにちは、Androidエンジニアの@morux2です。スタディサプリ小学・中学講座ではデータ通信にGraphQLを採用しており、開発者が信頼したクエリのみを処理する仕組みとしてTrusted Documentsを実装しています。この度、従来のPersisted QueryからSigned Queryという新しいTrusted Documentsの実装パターンに移行したので、Android側の実装についてご紹介できればと思います。

https://blog.studysapuri.jp/entry/2024/5/27/signed-query-android

# ViewModel 廃止検討

# ViewModel 廃止のモチベーション

画面の情報を1度のクエリで過不足なく取得できる場合, クエリの呼び出しが ViewModel と ApolloWrapper で二重にラップされてしまい, 冗長になる

```
fun fetchContent() {  morux2
    apolloWrapper.fetchMainScreenData().toLce().onEach { lce ->
        _viewState.update {
            it.copy(
                isLoading = lce.isLoading,
                throwable = lce.getThrowableIfError(),
                content = lce.getDataIfContent()
            )
        }
    }.launchIn(viewModelScope)
}
```

```
fun fetchMainScreenData(): Flow<MainScreenQuery.Data> {
    return apolloClient.query(MainScreenQuery())
        .toThrowableFlow()
}
```

# ViewModel 廃止のモチベーション

Apollo から取得した値を View に流すだけのテストは本質的には不要であり
VRT (Visual Regression Test) や E2E テストを拡充させる方が効果的である

```
@Test  new *
fun 画面が描画できること() {
    val viewModel = viewModelFactory(
        apolloWrapper = mockk {
            every { fetchMainScreenData() } returns flowOf(dummyContent)
        }
    )

    viewModel.fetchContent()

    viewModel.viewState.value shouldBe MainViewModel.ViewState(
        isLoading = false,
        throwable = null,
        content = dummyContent,
    )
}
```

# Apollo 3.8.0 で導入された toState() 関数

```
@OptIn(ApolloExperimental::class)  new *
@Composable
fun MainScreen(
    apolloClient: ApolloClient,
) {
    val state by apolloClient.query(MainScreenQuery()).toState()

    MainScreenContent(
        modifier = Modifier.fillMaxSize(),
        mainScreenState = state,
    )
}
```

https://github.com/apollographql/apollo-kotlin/releases/tag/v3.8.0

# Apollo 3.8.0 で導入された toState() 関数

```
@OptIn(ApolloExperimental::class)  new *
@Composable
fun MainScreenContent(
    mainScreenState: ApolloResponse<MainScreenQuery.Data>?,
    modifier: Modifier = Modifier
) {
    Scaffold(modifier = modifier) { contentPadding ->
        mainScreenState?.data?.let { content ->
            LazyColumn(
                modifier = Modifier
                    .padding(contentPadding)
                    .padding(horizontal = 8.dp),
                contentPadding = PaddingValues(vertical = 16.dp),
                verticalArrangement = Arrangement.spacedBy(16.dp)
            ) {
                // (省略) content を表示する
            }
        }
        if (mainScreenState?.hasErrors() == true || mainScreenState?.exception != null) {
            ErrorAlertDialog(
                throwable = mainScreenState.exception?.cause
                    ?: GraphQlServerException(errors = mainScreenState.errors!!),
                refech = {}, // refech が用意されていない
            )
        }
        if (mainScreenState == null) {
            Box(modifier = Modifier.fillMaxSize()) {
                CircularProgressIndicator(
                    modifier = Modifier.align(Alignment.Center)
                )
            }
```

# Apollo 3.8.0 で導入された toState() 関数

```
@OptIn(ApolloExperimental::class)  new *
@Composable
fun MainScreenContent(
    mainScreenState: ApolloResponse<MainScreenQuery.Data>?,
    modifier: Modifier = Modifier
) {
    Scaffold(modifier = modifier) { contentPadding ->
        mainScreenState?.data?.let { content ->
            LazyColumn(
                modifier = Modifier
                    .padding(contentPadding)
                    .padding(horizontal = 8.dp),
                contentPadding = PaddingValues(vertical = 16.dp),
                verticalArrangement = Arrangement.spacedBy(16.dp)
            ) {
                // (省略) content を表示する
            }
        }
        if (mainScreenState?.hasErrors() == true || mainScreenSt
            ErrorAlertDialog(
                throwable = mainScreenState.exception?.cause
                    ?: GraphQlServerException(errors = mainScree
                refech = {}, // refech が用意されていない
            )
        }
        if (mainScreenState == null) {
            Box(modifier = Modifier.fillMaxSize()) {
                CircularProgressIndicator(
                    modifier = Modifier.align(Alignment.Center)
                )
            }
```

refetch の機構が用意されていない

# toState() 関数を参考に, refetch 可能な関数を自作する

```
@Composable  new *
fun MainScreen(
    apolloClient: ApolloClient,
) {
    val coroutineScope = rememberCoroutineScope()
    val mainScreenQuery = apolloClient.refetchableQuery(query = MainScreenQuery())
    val mainLce by mainScreenQuery.toLce()

    LifecycleEventEffect(Lifecycle.Event.ON_RESUME) {
        coroutineScope.launch {
            mainScreenQuery.load()
        }
    }

    MainScreenContent(
        modifier = Modifier.fillMaxSize(),
        lce = mainLce,
        refetch = {
            coroutineScope.launch {
                mainScreenQuery.load()
            }
        },
    )
}
```

# toState() 関数を参考に, refetch 可能な関数を自作する

```
@Composable  new *
fun MainScreenContent(
    lce: Lce<MainScreenQuery.Data>,
    refetch: () -> Unit,
    modifier: Modifier = Modifier
) {
    Scaffold(modifier = modifier) { contentPadding ->
        lce.getDataIfContent()?.let { content ->
            LazyColumn(
                modifier = Modifier
                    .padding(contentPadding)
                    .padding(horizontal = 8.dp),
                contentPadding = PaddingValues(vertical = 16.dp),
                verticalArrangement = Arrangement.spacedBy(16.dp)
            ) {
                // (省略) content を表示する
            }
        }
        lce.getThrowableIfError()?.let { throwable ->
            ErrorAlertDialog(
                throwable = throwable,
                refech = refetch,
            )
        }

        if (lce.isLoading) {
            Box(modifier = Modifier.fillMaxSize()) {
                CircularProgressIndicator(
                    modifier = Modifier.align(Alignment.Center)
                )
            }
```

# toState() 関数を参考に, refetch 可能な関数を自作する

```
// 拡張関数を生やすことで, remember ブロックの囲み忘れを防ぐ
@Composable
fun <T : Query.Data> ApolloClient.refetchableQuery(query: Query<T>): RefetchableQuery<T> {
    // recomposition のたびに, RefetchableQuery (refechTrigger) が生成されてしまい意図しない挙動になる
    return remember {
        RefetchableQuery(
            apolloClient = this,
            query = query
        )
    }
}
```

**共通実装として定義することで,**
**画面ごとに recomposition を**
**考慮する手間をなくす**

# toState() 関数を参考に, refetch 可能な関数を自作する

```
@OptIn(ExperimentalCoroutinesApi::class) new *
data class RefetchableQuery<T : Query.Data>(
    val apolloClient: ApolloClient,
    val query: Query<T>,
) {
    // 購読開始前に値が流れてしまうケースがあるので, replay を 1 に設定する
    private val refechTrigger = MutableSharedFlow<Unit>(replay = 1)

    @Composable new *
    fun toLce(): State<Lce<T>?> {
        val responseFlow = remember {
            refechTrigger.flatMapLatest {
                apolloClient.query(query = query)
                    .toThrowableFlow()
                    .onEach {
                        // refechTrigger 契機で Flow が一度発火したら, 以降は再発火しないようにする
                        refechTrigger.resetReplayCache()
                    }
                    .toLce()
            }
        }
        // 初期値を null にしておくことで, toLce 呼び出し時点でローディングが流れてしまうことを避ける
        return responseFlow.collectAsStateWithLifecycle(initialValue = null)
    }

    suspend fun load() { new *
        refechTrigger.emit(Unit)
    }
}
```

**toLce() を呼び出すことで,**
**LCE の状態を購読できる**

**load() を呼び出すことで,**
**Query を実行して値を流す**

# ViewModel を廃止した実装の所感

- 😊 ボイラープレートコードが減る
- 🤨 Recomposition を意識して Query / Mutation を呼び出す必要がある
    - 画面ごとに Recomposition を意識することになると, 実装難易度とコストが高いので, 適切な共通実装に落とし込むことがポイントになる