Qusion

Kotlin in Consuming GraphQL on Android

Miki Mitevski

Android Developer @Qusion

EPISODE I

Apollo - The coroutine way

EPISODE II

A New Hope - Channels

APOLLO

APOLLO

apollographql / apollo-android

Watch 93 | Unstar 2k | Fork 369

Code | Issues 75 | Pull requests 7 | Actions | Wiki | Security | Insights

A strongly-typed, caching GraphQL client for Android and the JVM

graphql-client | android | apollographql | graphql

984 commits | 11 branches | 0 packages | 27 releases | 88 contributors | MIT

Branch: master | New pull request | Create new file | Upload files | Find file | Clone or download

sav007 Set theme jekyll-theme-minimal — Latest commit da39d5e 17 hours ago

.buildscript	Introduce a global "composite" project for samples and integration te...	last month
.github	Update issue templates	6 days ago
.idea/codeStyles	Add subscription state listener (#1691)	29 days ago
apollo-android-support	trying to revive the connectedTests (#1791)	18 hours ago
apollo-api	Generate function to parse operation response (#1761)	8 days ago
apollo-compiler	Generate computed properties to access inline fragment instances (#1780)	5 days ago
apollo-coroutines-support	move build scripts to kotlin (#1654)	last month

Schema.json

```json
{
  "__schema": {
    "queryType": {
      "name": "Query"
    },
    "mutationType": {
      "name": "Mutation"
    },
    "types": [
      {
        "fields":
          ...
          {
            "name": "member",
            "description": null,
            "args": [],
            "type": {
              "kind": "OBJECT",
              "name": "Member",
              "ofType": null
            },
            "isDeprecated": false,
            "deprecationReason": null
          }
          ...
      }
     ]
    ...
}
```

Query.graphql

```graphql
query FeedQuery($type: FeedType!, $limit: Int!) {
  feedEntries: feed(type: $type, limit: $limit) {
    id
    repository {
      name
    }
    postedBy {
      login
    }
  }
}
```

Usage

```java
apolloClient().query(feedQuery)
        .enqueue(new ApolloCallback<>(new ApolloCall.Callback<FeedQuery.Data>() {
          @Override public void onResponse(@NotNull Response<FeedQuery.Data> response) {
            Log.i(TAG, response.toString());
          }

          @Override public void onFailure(@NotNull ApolloException e) {
            Log.e(TAG, e.getMessage(), e);
          }
        }, uiHandler));
```

EPISODE I

Apollo - The Coroutine Way

The Problem?

```
01 final HeroAndFriendsNames heroAndFriendsQuery = HeroAndFriendsNames.builder()
02     .episode(NEWHOPE)
03     .build();
04
05 apolloClient().query(heroAndFriendsQuery)
06     .enqueue(new ApolloCallback<>(
07         new ApolloCall.Callback<HeroAndFriendsNames.Data>() {
08       @Override
09       public void onResponse(
10           @NotNull Response<HeroAndFriendsNames.Data> response) {
11         Log.i(TAG, response.toString());
12       }
13
14       @Override
15       public void onFailure(@NotNull ApolloException e) {
16         Log.e(TAG, e.getMessage(), e);
17       }
18     }, uiHandler));
19 }
```

```
01 final HeroAndFriendsNames heroAndFriendsQuery = HeroAndFriendsNames.builder()
02     .episode(NEWHOPE)
03     .build();
04
05 apolloClient().query(heroAndFriendsQuery)
06     .enqueue(new ApolloCallback<>(
07         new ApolloCall.Callback<HeroAndFriendsNames.Data>() {
08       @Override
09       public void onResponse(
10           @NotNull Response<HeroAndFriendsNames.Data> response) {
11         Log.i(TAG, response.toString());
12       }
13
14       @Override
15       public void onFailure(@NotNull ApolloException e) {
16         Log.e(TAG, e.getMessage(), e);
17       }
18     }, uiHandler));
19 }
```

Traditional callbacks :(

THE SOLUTION

Coroutines

UI / ViewModel

↓

Repository / Domain

↓

Apollo Service

```
01 // in :repository module
02 class MemberRepository(private val apolloService: ApolloService) {
03
04     ...
05
06     suspend fun getMemberData(): MemberQuery.Data {
07         val response = apolloService.getClient()
08             .query(MemberQuery.builder().build())
09             .toDeferred().await()
10
11         return response
12     }
13
14     ...
15 }
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01 // in :repository module
02 class MemberRepository(private val apolloService: ApolloService) {
03
04     ...
05
06     suspend fun getMemberData(): MemberQuery.Data {
07         val response = apolloService.getClient()
08             .query(MemberQuery.builder().build())
09             .toDeferred().await()
10
11         return response
12     }
13
14     ...
15 }
```

UI / ViewModel

Repository / Domain

Apollo Service

ApolloClient wrapper

```
01 // in :apollo-service module
02 class ApolloService(private val context: Context) {
03
04     @Volatile
05     private var client: ApolloClient? = null
06
07     fun getClient(): ApolloClient {
08         return client ?: synchronized(this) {
09             client ?: buildApolloClient().also {
10                 client = it
11             }
12         }
13     }
14
15     private fun buildApolloClient(): ApolloClient {
16
17         ...
18
19         return ApolloClient.builder()
20             .serverUrl(BuildConfig.API_URL)
21             .okHttpClient(okHttpClient)
22             .normalizedCache(cacheFactory, resolver)
23             .build()
24     }
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01 // in :repository module
02 class MemberRepository(private val apolloService: ApolloService) {
03
04     ...
05
06     suspend fun getMemberData(): MemberQuery.Data {
07         val response = apolloService.getClient()
08             .query(MemberQuery.builder().build())
09             .toDeferred().await()
10
11         return response
12     }
13
14     ...
15 }
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01 // in :repository module
02 class MemberRepository(private val apolloService: ApolloService) {
03
04     ...
05
06     suspend fun getMemberData(): MemberQuery.Data {
07         val response = apolloService.getClient()
08             .query(MemberQuery.builder().build())
09             .toDeferred().await()
10
11         return response
12     }
13
14     ...
15 }
```

UI / ViewModel

Repository / Domain

Apollo Service

Awaits for completion

```
01 //From apollo-android library source code
02 fun <T> ApolloCall<T>.toDeferred(): Deferred<Response<T>> {
03     val deferred = CompletableDeferred<Response<T>>()
04
05     deferred.invokeOnCompletion {
06       if (deferred.isCancelled) {
07         cancel()
08       }
09     }
10     enqueue(object : ApolloCall.Callback<T>() {
11       override fun onResponse(response: Response<T>) {
12         deferred.complete(response)
13       }
14
15       override fun onFailure(e: ApolloException) {
16         deferred.completeExceptionally(e)
17       }
18     })
19
20     return deferred
21 }
```

```
01 //From apollo-android library source code
02 fun <T> ApolloCall<T>.toDeferred(): Deferred<Response<T>> {
03     val deferred = CompletableDeferred<Response<T>>()
04
05     deferred.invokeOnCompletion {
06       if (deferred.isCancelled) {
07         cancel()
08       }
09     }
10     enqueue(object : ApolloCall.Callback<T>() {
11       override fun onResponse(response: Response<T>) {
12         deferred.complete(response)
13       }
14
15       override fun onFailure(e: ApolloException) {
16         deferred.completeExceptionally(e)
17       }
18     })
19
20     return deferred
21 }
```

Creates a Deferred object

```
01 //From apollo-android library source code
02 fun <T> ApolloCall<T>.toDeferred(): Deferred<Response<T>> {
03     val deferred = CompletableDeferred<Response<T>>()
04
05     deferred.invokeOnCompletion {
06       if (deferred.isCancelled) {
07         cancel()
08       }
09     }
10     enqueue(object : ApolloCall.Callback<T>() {
11       override fun onResponse(response: Response<T>) {
12         deferred.complete(response)
13       }
14
15       override fun onFailure(e: ApolloException) {
16         deferred.completeExceptionally(e)
17       }
18     })
19
20     return deferred
21 }
```

Still a callback under the hood

```
01 // in :repository module
02 class MemberRepository(private val apolloService: ApolloService) {
03
04     ...
05
06     suspend fun getMemberData(): MemberQuery.Data {
07         val response = apolloService.getClient()
08             .query(MemberQuery.builder().build())
09             .toDeferred().await()
10
11         return response
12     }
13
14     ...
15 }
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01 // in :repository module
02 class MemberRepository(private val apolloService: ApolloService) {
03
04     ...
05
06     suspend fun getMemberData(): MemberQuery.Data {
07         val response = apolloService.getClient()
08             .query(MemberQuery.builder().build())
09             .toDeferred().await()
10
11         return response
12     }
13
14     ...
15 }
```

```
1 sealed class NetworkResult<out T : Any> {
2     data class Success<out T : Any>(val value: T) : NetworkResult<T>()
3     data class Error(val cause: Exception? = null) : NetworkResult<Nothing>()
4 }
```

```
01 // in :repository module
02 class MemberRepository(private val apolloService: ApolloService) {
03
04     ...
05
06     suspend fun getMemberData(): NetworkResult<MemberQuery.Data> {
07         try {
08             val response = apolloService.getClient()
09                 .query(MemberQuery.builder().build())
10                 .toDeferred().await()
11
12             return NetworkResult.Success(response)
13         }
14         catch(e: ApolloNetworkException) {
15             return NetworkResult.Error(e)
16         }
17     }
18
19     ...
20 }
```

```
1 sealed class NetworkResult<out T : Any> {
2     data class Success<out T : Any>(val value: T) : NetworkResult<T>()
3     data class Error(val cause: Exception? = null) : NetworkResult<Nothing>()
4 }
```

```
01  //In ApolloServiceImpl
02  override suspend fun <D : Operation.Data, T : Operation.Data, V : Operation.Variables>
query(
03      query: Query<D, T, V>,
04      cachePolicy: HttpCachePolicy.Policy,
05      responseFetcher: ResponseFetcher
06  ): NetworkResult<T> {
07      try {
08          val response = getClient().query(query)
09              .httpCachePolicy(cachePolicy).responseFetcher(responseFetcher)
10              .toDeferred().await()
11
12          if (response.hasErrors()) {
13              val error = response.errors().first()
14              return NetworkResult.Error(
15                  cause = BusinessException(error.message())
16              )
17          }
18
19          return NetworkResult.Success(response.data())
20      } catch (e: ApolloNetworkException) {
21          return NetworkResult.Error(e)
22      }
23  }
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01  //In ApolloServiceImpl
02  override suspend fun <D : Operation.Data, T : Operation.Data, V : Operation.Variables>
query(
03      query: Query<D, T, V>,
04      cachePolicy: HttpCachePolicy.Policy,
05      responseFetcher: ResponseFetcher
06  ): NetworkResult<T> {
07      try {
08          val response = getClient().query(query)
09              .httpCachePolicy(cachePolicy).responseFetcher(responseFetcher)
10              .toDeferred().await()
11
12          if (response.hasErrors()) {
13              val error = response.errors().first()
14              return NetworkResult.Error(
15                  cause = BusinessException(error.message())
16              )
17          }
18
19          return NetworkResult.Success(response.data())
20      } catch (e: ApolloNetworkException) {
21          return NetworkResult.Error(e)
22      }
23  }
```

UI / ViewModel

Repository / Domain

Apollo Service

Represents the fetch strategy

```
//In ApolloServiceImpl
override suspend fun <D : Operation.Data, T : Operation.Data, V : Operation.Variables>
query(
    query: Query<D, T, V>,
    cachePolicy: HttpCachePolicy.Policy,
    responseFetcher: ResponseFetcher
): NetworkResult<T> {
    try {
        val response = getClient().query(query)
                                                    ponseFetcher)
```

```
/**
 * Represents different fetch strategies for http request / response cache
 */
public enum FetchStrategy {
  /**
   * Signals the apollo client to fetch the GraphQL query response from the http cache <b>only</b>.
   */
  CACHE_ONLY,
  /**
   * Signals the apollo client to fetch the GraphQL query response from the network <b>only</b>.
   */
  NETWORK_ONLY,
  /**
   * Signals the apollo client to first fetch the GraphQL query response from the http cache. If it's not present in
   * the cache response is fetched from the network.
   */
  CACHE_FIRST,
  /**
   * Signals the apollo client to first fetch the GraphQL query response from the network. If it fails then fetch the
   * response from the http cache.
   */
  NETWORK_FIRST
}
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01 //In ApolloServiceImpl
02 override suspend fun <D : Operation.Data, T : Operation.Data, V : Operation.Variables> query(
03     query: Query<D, T, V>,
04     cachePolicy: HttpCachePolicy.Policy,
05     responseFetcher: ResponseFetcher
06 ): NetworkResult<T> {
07     try {
08         val response = getClient().query(query)
09             .httpCachePolicy(cachePolicy).responseFetcher(responseFetcher)
10             .toDeferred().await()
11
12         if (response.hasErrors()) {
13             val error = response.errors().first()
14             return NetworkResult.Error(
15                 cause = BusinessException(error.message())
16             )
17         }
18
19         return NetworkResult.Success(response.data())
20     } catch (e: ApolloNetworkException) {
21         return NetworkResult.Error(e)
22     }
23 }
```

UI / ViewModel

Repository / Domain

Apollo Service

```fun
//In ApolloServiceImpl
override suspend fun <D : Operation.Data, T : Operation.Data, V : Operation.Variables> query(
    query: Query<D, T, V>,
    cachePolicy: HttpCachePolicy.Policy,
    responseFetcher: ResponseFetcher
): NetworkResult<T> {
    try {
        val response = getClient().query(query)
            .httpCachePolicy(cachePolicy).responseFetcher(responseFetcher)
            .toDeferred().await()

        if (response.hasErrors()) {
            val error = response.errors().first()
            return NetworkResult.Error(
                cause = BusinessException(error.message())
            )
        }

        return NetworkResult.Success(response.data())
    } catch (e: ApolloNetworkException) {
        return NetworkResult.Error(e)
    }
}
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01
02 class MemberRepository(private val apolloService: ApolloService) {
03
04     ...
05
06     suspend fun getMember(
07         cachePolicy: HttpCachePolicy.Policy,
08         responseFetcher: ResponseFetcher
09     ): NetworkResult<MemberQuery.Data> {
10
11         val memberQuery = MemberQuery.builder().build()
12
13         return apolloService.query(
14             query = memberQuery,
15             cachePolicy = cachePolicy,
16             responseFetcher = responseFetcher
17         )
18     }
19
20     ...
21 }
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01  class MemberViewModel(private val repository: MemberRepository) : ViewModel() {
02
03      private val _memberData: MutableLiveData<MemberQuery.Member> = MutableLiveData()
04      val memberData: LiveData<MemberQuery.Member> = _memberData
05
06      fun getMember() {
07
08         viewModelScope.launch(Dispatchers.IO) {
09
10             when (val response = repository.getMember(
11                 HttpCachePolicy.CACHE_FIRST,
12                 ApolloResponseFetchers.CACHE_FIRST
13              )) {
14                 is NetworkResult.Success -> {
15                     _memberData.postValue(response.value.member())
16                 }
17                 is NetworkResult.Error -> {
18                     //Handle error
19                 }
20             }
21         }
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01 class MemberViewModel(private val repository: MemberRepository) : ViewModel() {
02
03     private val _memberData: MutableLiveData<MemberQuery.Member> = MutableLiveData()
04     val memberData: LiveData<MemberQuery.Member> = _memberData
05
06     fun getMember() {
07
08         viewModelScope.launch(Dispatchers.IO) {
09
10             when (val response = repository.getMember(
11                 HttpCachePolicy.CACHE_FIRST,
12                 ApolloResponseFetchers.CACHE_FIRST
13             )) {
14                 is NetworkResult.Success -> {
15                     _memberData.postValue(response.value.member())
16                 }
17                 is NetworkResult.Error -> {
18                     //Handle error
19                 }
20             }
21         }
```

UI / ViewModel

Repository / Domain

Apollo Service

Expose some LiveData

```
01  class MemberViewModel(private val repository: MemberRepository) : ViewModel() {
02
03      private val _memberData: MutableLiveData<MemberQuery.Member> = MutableLiveData()
04      val memberData: LiveData<MemberQuery.Member> = _memberData
05
06      fun getMember() {
07
08          viewModelScope.launch(Dispatchers.IO) {
09
10              when (val response = repository.getMember(
11                  HttpCachePolicy.CACHE_FIRST,
12                  ApolloResponseFetchers.CACHE_FIRST
13               )) {
14                  is NetworkResult.Success -> {
15                      _memberData.postValue(response.value.member())
16                  }
17                  is NetworkResult.Error -> {
18                      //Handle error
19                  }
20              }
21          }
```

UI / ViewModel

Repository / Domain

Apollo Service

Tied to the lifecycle of the ViewModel

```
03      private val _memberData: MutableLiveData<MemberQuery.Member> = MutableLiveData()
04      val memberData: LiveData<MemberQuery.Member> = _memberData
05
06      fun getMember() {
07
08         viewModelScope.launch(Dispatchers.IO) {
09
10            when (val response = repository.getMember(
11                HttpCachePolicy.CACHE_FIRST,
12                ApolloResponseFetchers.CACHE_FIRST
13             )) {
14                is NetworkResult.Success -> {
15                    _memberData.postValue(response.value.member())
16                }
17                is NetworkResult.Error -> {
18                    //Handle error
19                }
20            }
21         }
22      }
23  }
```

UI / ViewModel

Repository / Domain

Apollo Service

How sealed classes are useful

Recap

EPISODE II

A New Hope - Channels

The Problem?

THE SOLUTION

Channels & Flows

```
01  //In ApolloServiceImpl
02  override suspend fun <D : Operation.Data, T : Operation.Data, V :
Operation.Variables> openQuery(
03      query: Query<D, T, V>
04  ): Flow<NetworkResult<T>> {
05      return getClient().query(query)
06             .responseFetcher(ApolloResponseFetchers.CACHE_AND_NETWORK)
07             .toNetworkFlow()
08  }
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01  //In ApolloServiceImpl
02  override suspend fun <D : Operation.Data, T : Operation.Data, V :
Operation.Variables> openQuery(
03      query: Query<D, T, V>
04  ): Flow<NetworkResult<T>> {
05      return getClient().query(query)
06              .responseFetcher(ApolloResponseFetchers.CACHE_AND_NETWORK)
07              .toNetworkFlow()
08  }
```

UI / ViewModel

Repository / Domain

Apollo Service

Allows to get a "flow" of data

```
01  //In ApolloServiceImpl
02  override suspend fun <D : Operation.Data, T : Operation.Data, V :
Operation.Variables> openQuery(
03      query: Query<D, T, V>
04  ): Flow<NetworkResult<T>> {
05      return getClient().query(query)
06              .responseFetcher(ApolloResponseFetchers.CACHE_AND_NETWORK)
07              .toNetworkFlow()
08  }
```

UI / ViewModel

Repository / Domain

Apollo Service

An ApolloCall extension function

```
01 fun <T : Any> ApolloCall<T>.toNetworkFlow() = flow {
02     val channel = Channel<Response<T>>(Channel.CONFLATED)
03
04     enqueue(ChannelCallback(channel = channel))
05     try {
06         for (it in channel) {
07             if (!it.hasErrors()) {
08                 emit(NetworkResult.Success(it.data()))
09             }
10             else {
11                 val error = response.errors().first()
12                 emit(
13                     NetworkResult.Error(BusinessException(error.message()))
14                 )
15             }
16         }
17     } catch (e: ApolloNetworkException) {
18         emit(NetworkResult.Error(e))
19     } finally {
20         cancel()
21     }
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01 fun <T : Any> ApolloCall<T>.toNetworkFlow() = flow {
02     val channel = Channel<Response<T>>(Channel.CONFLATED)
03
04     enqueue(ChannelCallback(channel = channel))
05     try {
06         for (it in channel) {
07             if (!it.hasErrors()) {
08                 emit(NetworkResult.Success(it.data()))
09             }
10             else {
11                 val error = response.errors().first()
12                 emit(
13                     NetworkResult.Error(BusinessException(error.message()))
14                 )
15             }
16         }
17     } catch (e: ApolloNetworkException) {
18         emit(NetworkResult.Error(e))
19     } finally {
20         cancel()
21     }
```

UI / ViewModel

Repository / Domain

Apollo Service

Cold streams

```
01 fun <T : Any> ApolloCall<T>.toNetworkFlow() = flow {
02     val channel = Channel<Response<T>>(Channel.CONFLATED)
03
04     enqueue(ChannelCallback(channel = channel))
05     try {
06         for (it in channel) {
07             if (!it.hasErrors()) {
08                 emit(NetworkResult.Success(it.data()))
09             }
10             else {
11                 val error = response.errors().first()
12                 emit(
13                     NetworkResult.Error(BusinessException(error.message()))
14                 )
15             }
16         }
17     } catch (e: ApolloNetworkException) {
18         emit(NetworkResult.Error(e))
19     } finally {
20         cancel()
21     }
```

UI / ViewModel

Repository / Domain

Apollo Service

Creates a channel where the response will be passed through

```
01 fun <T : Any> ApolloCall<T>.toNetworkFlow() = flow {
02     val channel = Channel<Response<T>>(Channel.CONFLATED)
03
04     enqueue(ChannelCallback(channel = channel))
05     try {
06         for (it in channel) {
07             if (!it.hasErrors()) {
08                 emit(NetworkResult.Success(it.data()))
09             }
10             else {
11                 val error = response.errors().first()
12                 emit(
13                     NetworkResult.Error(BusinessException(error.message()))
14                 )
15             }
16         }
17     } catch (e: ApolloNetworkException) {
18         emit(NetworkResult.Error(e))
19     } finally {
20         cancel()
21     }
```

UI / ViewModel

Repository / Domain

Apollo Service

Extends ApolloCall.Callback and sets up the channel

```
01 fun <T : Any> ApolloCall<T>.toNetworkFlow() = flow {
02     val channel = Channel<Response<T>>(Channel.CONFLATED)
03
04     enqueue(ChannelCallback(channel = channel))
05     try {
06         for (it in channel) {
07             if (!it.hasErrors()) {
08                 emit(NetworkResult.Success(it.data()))
09
10
11
12
13
14
15
16         }
17     } catch
18         emit
19     } finally
20         canc
21     }
```

```
1 private class ChannelCallback<T>(val channel: Channel<Response<T>>) : ApolloCall.Callback<T>() {
2
3     override fun onResponse(response: Response<T>) {
4         channel.offer(response)
5     }
6
7     override fun onFailure(e: ApolloException) {
8         channel.close(e)
9     }
10
11    override fun onStatusEvent(event: ApolloCall.StatusEvent) {
12         if (event == ApolloCall.StatusEvent.COMPLETED) {
13            channel.close()
14         }
15    }
16 }
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01  fun <T : Any> ApolloCall<T>.toNetworkFlow() = flow {
02      val channel = Channel<Response<T>>(Channel.CONFLATED)
03
04      enqueue(ChannelCallback(channel = channel))
05      try {
06          for (it in channel) {
07              if (!it.hasErrors()) {
08                  emit(NetworkResult.Success(it.data()))
09              }
10              else {
11                  val error = response.errors().first()
12                  emit(
13                      NetworkResult.Error(BusinessException(error.message()))
14                  )
15              }
16          }
17      } catch (e: ApolloNetworkException) {
18          emit(NetworkResult.Error(e))
19      } finally {
20          cancel()
21      }
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01 fun <T : Any> ApolloCall<T>.toNetworkFlow() = flow {
02     val channel = Channel<Response<T>>(Channel.CONFLATED)
03
04     enqueue(ChannelCallback(channel = channel))
05     try {
06         for (it in channel) {
07             if (!it.hasErrors()) {
08                 emit(NetworkResult.Success(it.data()))
09             }
10             else {
11                 val error = response.errors().first()
12                 emit(
13                     NetworkResult.Error(BusinessException(error.message()))
14                 )
15             }
16         }
17     } catch (e: ApolloNetworkException) {
18         emit(NetworkResult.Error(e))
19     } finally {
20         cancel()
21     }
```

UI / ViewModel

Repository / Domain

Apollo Service

Emit should happen strictly in the dispatchers of the block in order to preserve the flow context

```
01 class MemberViewModel(private val repository: MemberRepository) : ViewModel() {
02     ...
03
04     val member: LiveData<MemberQuery.Member> =
05         liveData(viewModelScope.coroutineContext + Dispatchers.IO) {
06             repository.getMemberOpen()
07                 .collect {
08                     if (it is NetworkResult.Success) {
09                         emit(it.value.member())
10                     }
11                 }
12         }
13
14     ...
15 }
```

UI / ViewModel

Repository / Domain

Apollo Service

```
01 class MemberViewModel(private val repository: MemberRepository) : ViewModel() {
02     ...
03
04     val member: LiveData<MemberQuery.Member> =
05         liveData(viewModelScope.coroutineContext + Dispatchers.IO) {
06             repository.getMemberOpen()
07                 .collect {
08                     if (it is NetworkResult.Success) {
09                         emit(it.value.member())
10                     }
11                 }
12         }
13
14     ...
15 }
```

UI / ViewModel

Repository / Domain

Apollo Service

Starts executing when live data becomes active

```
01 class MemberViewModel(private val repository: MemberRepository) : ViewModel() {
02     ...
03
04     val member: LiveData<MemberQuery.Member> =
05         liveData(viewModelScope.coroutineContext + Dispatchers.IO) {
06             repository.getMemberOpen()
07                 .collect {
08                     if (it is NetworkResult.Success) {
09                         emit(it.value.member())
10                     }
11                 }
12         }
13
14     ...
15 }
```

UI / ViewModel

Repository / Domain

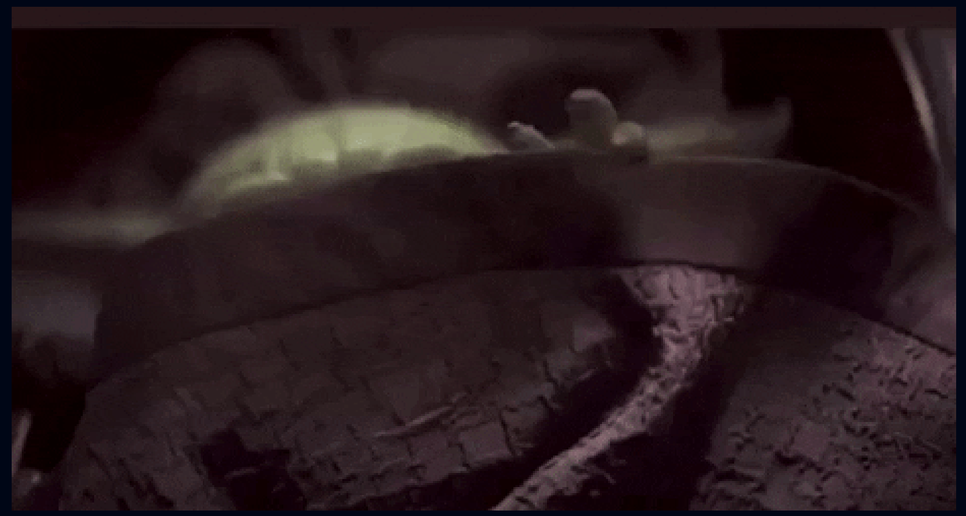
Apollo Service

Collect the values of the flow

Recap

Qusion

To be continued...



FOLLOW UP

qusion.com/blog