Taming UDF architectures - Real life cases

Укрощаем UDF архитектуры.

Igor Karenkov

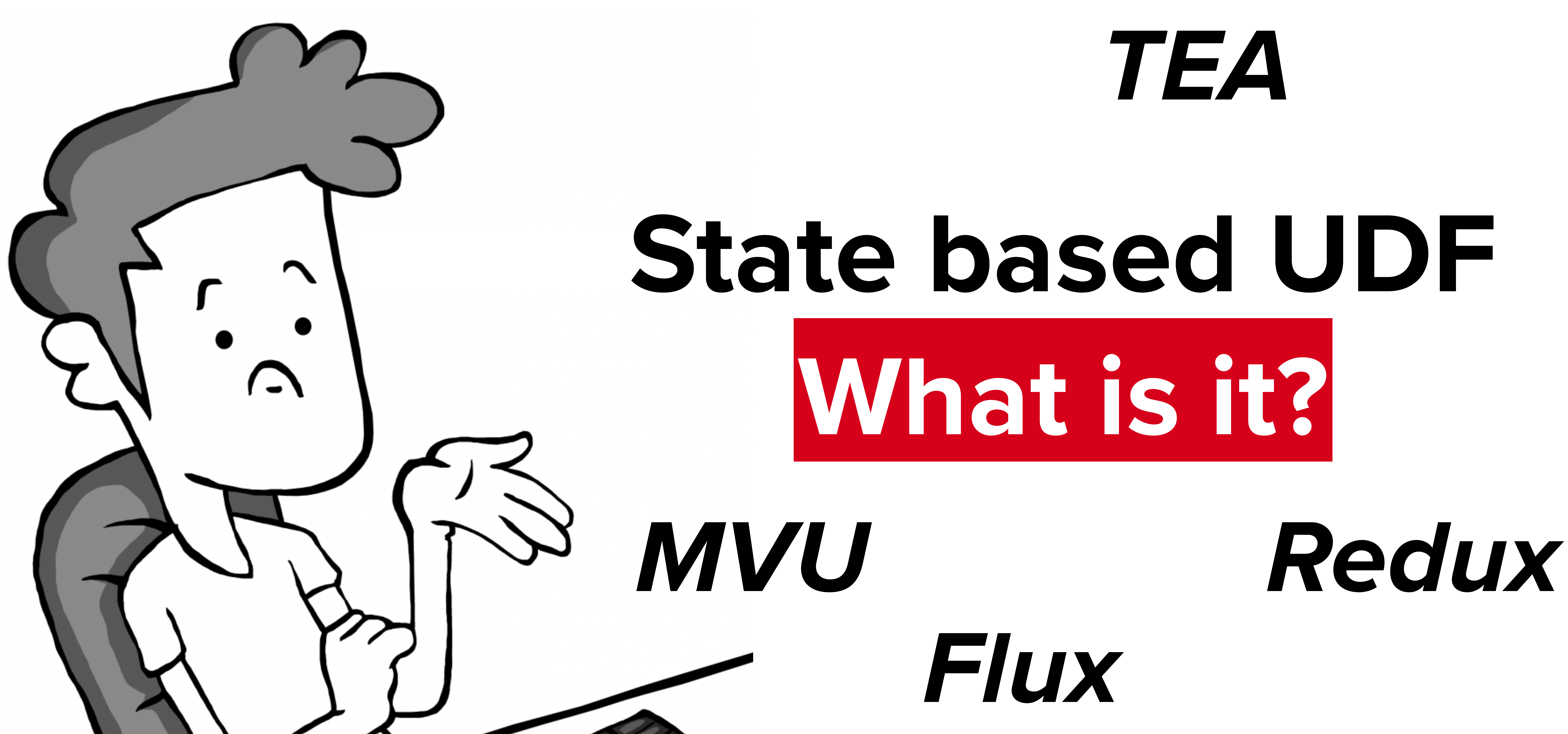
7 years in Android dev,
TeamLead mobile-core @ HH

Develop open source
(Modo & Kombucha-UDF)

Mentoring developers

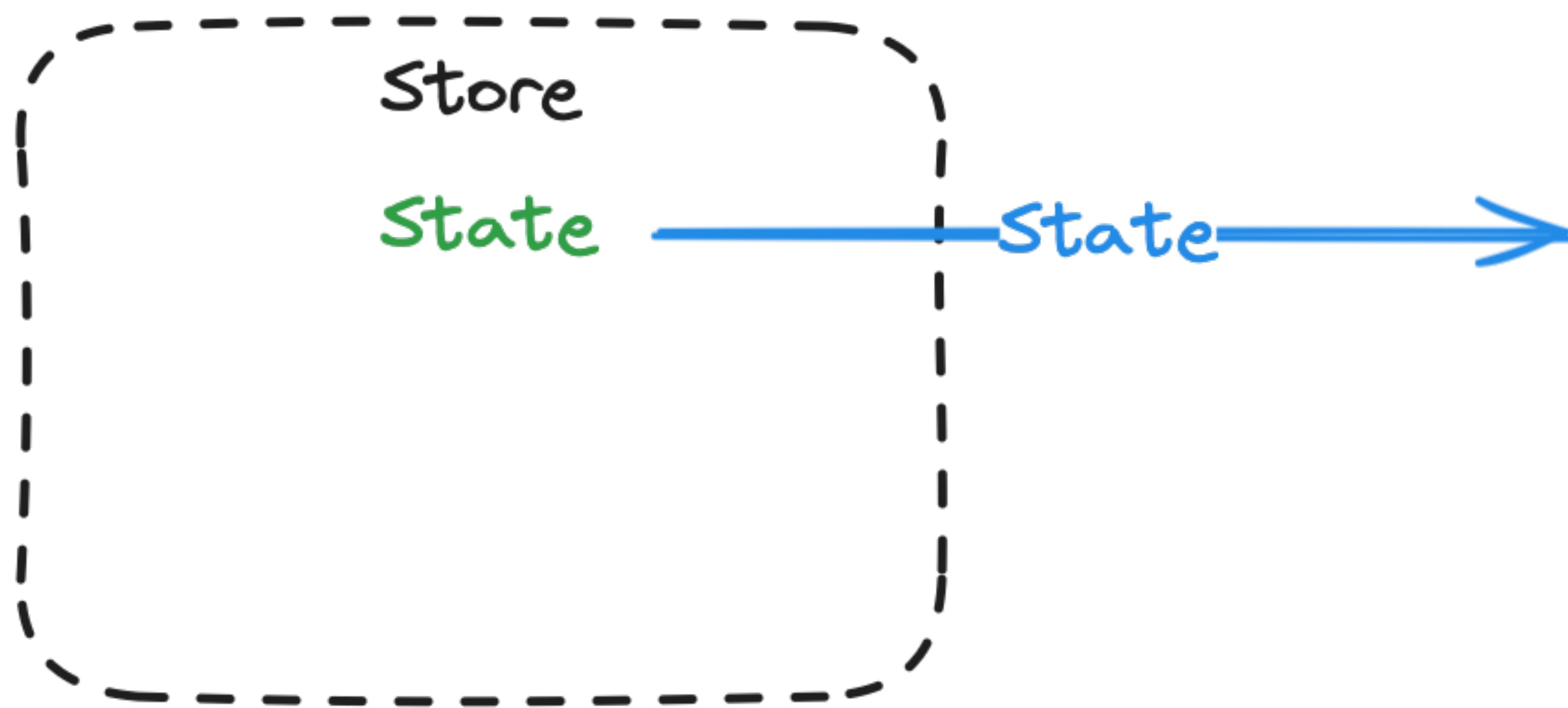
Rock climber

Mentoring -
https://getmentor.dev/mentor/igor-karenkov-1058


MVI
TEA
State based UDF
What is it?
MVU
Redux
Flux

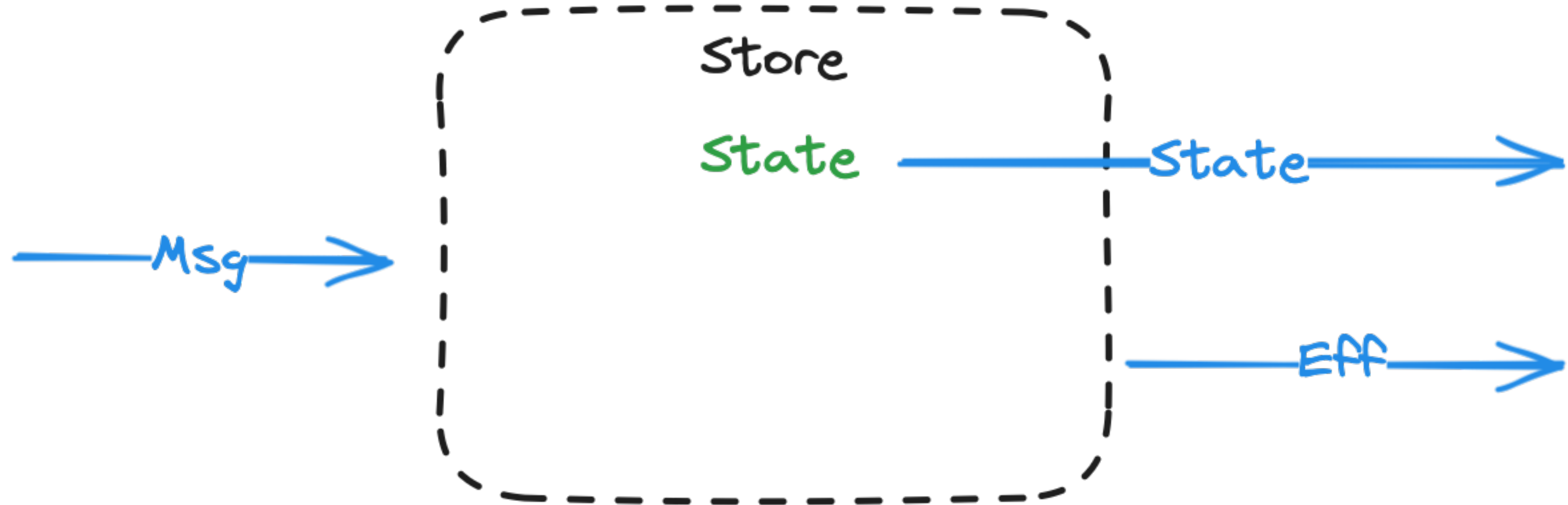
State-based UDF

State-based UDF

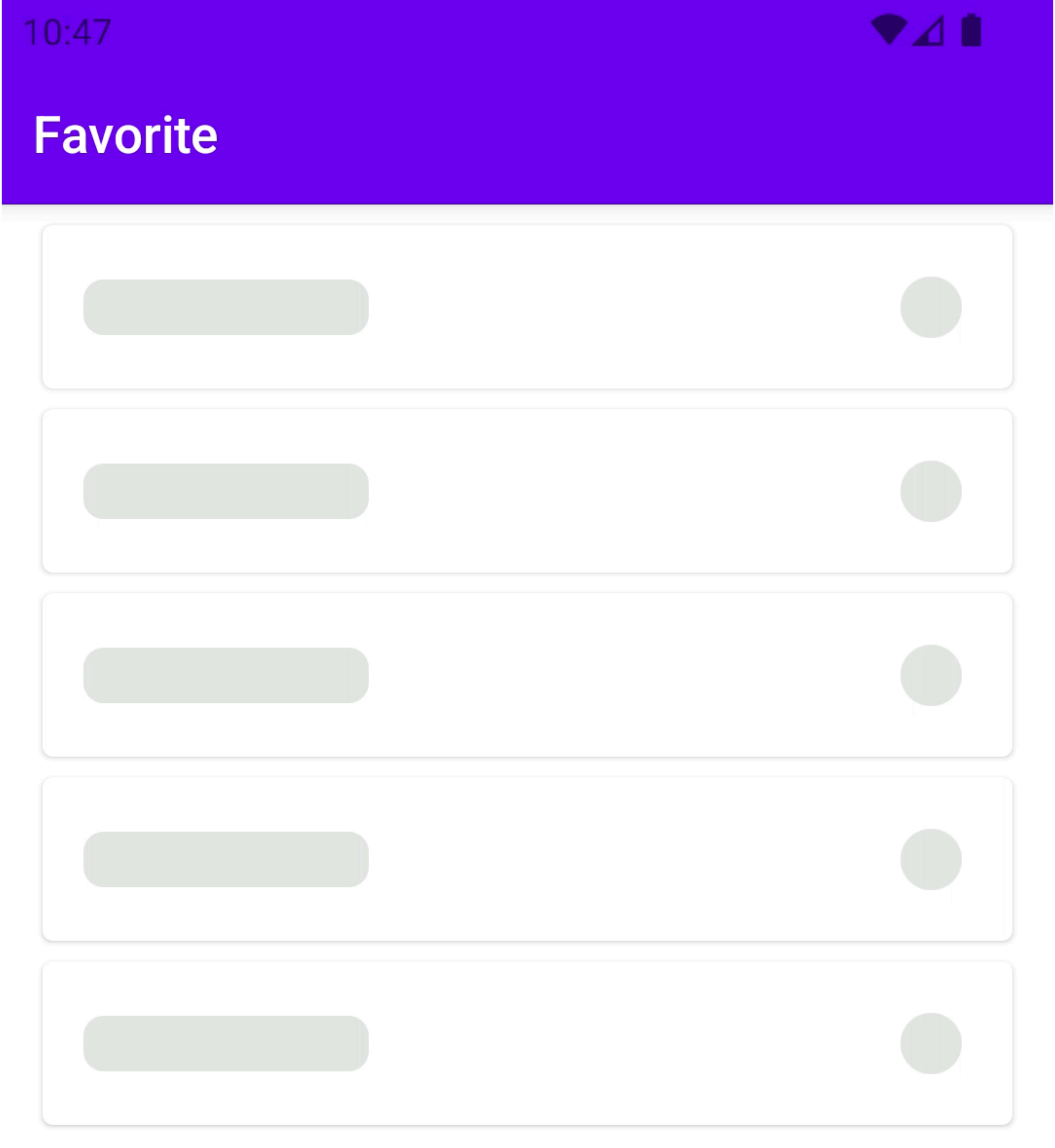


State-based UDF

State-based UDF



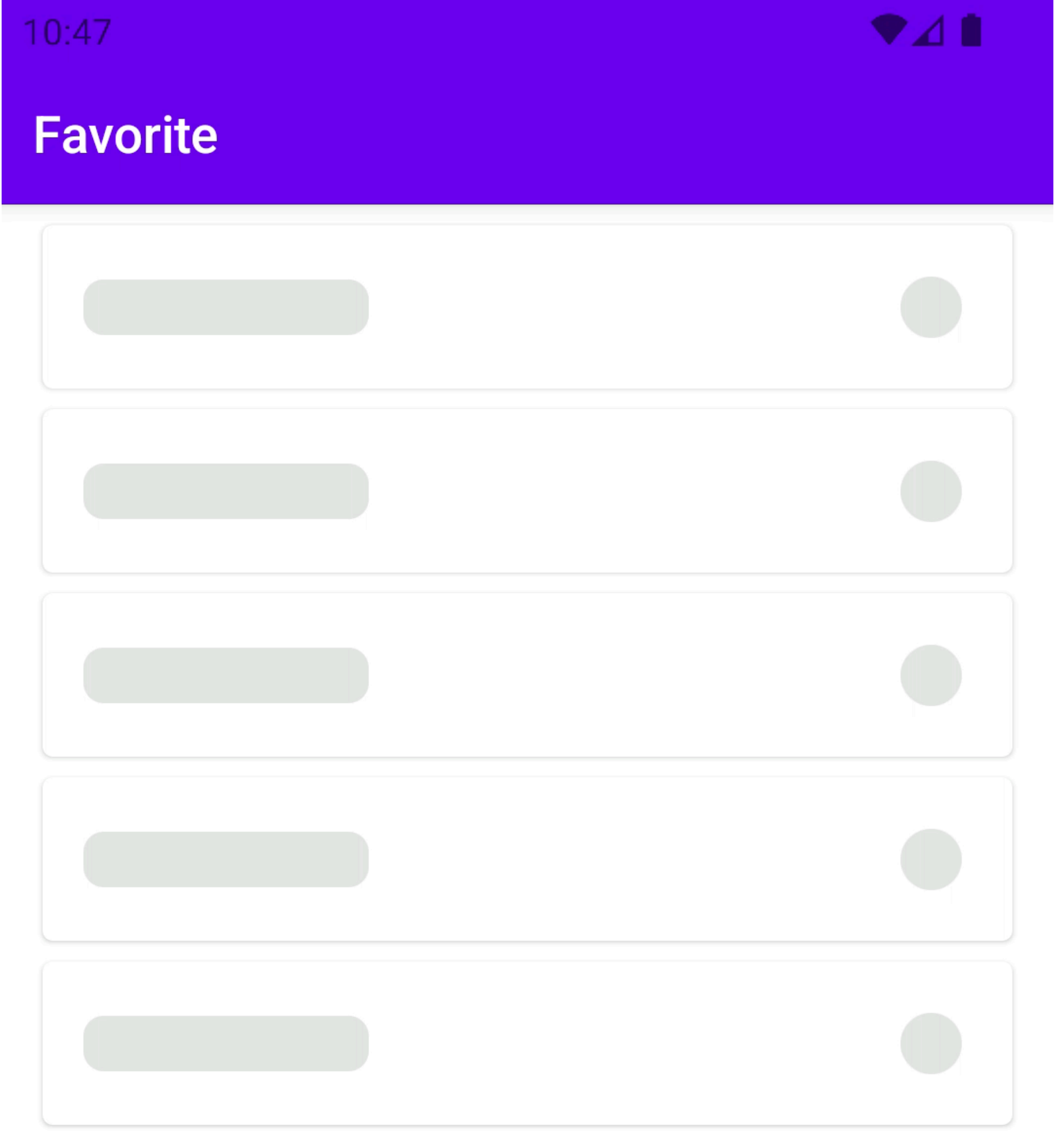
UDF sample: favorite list



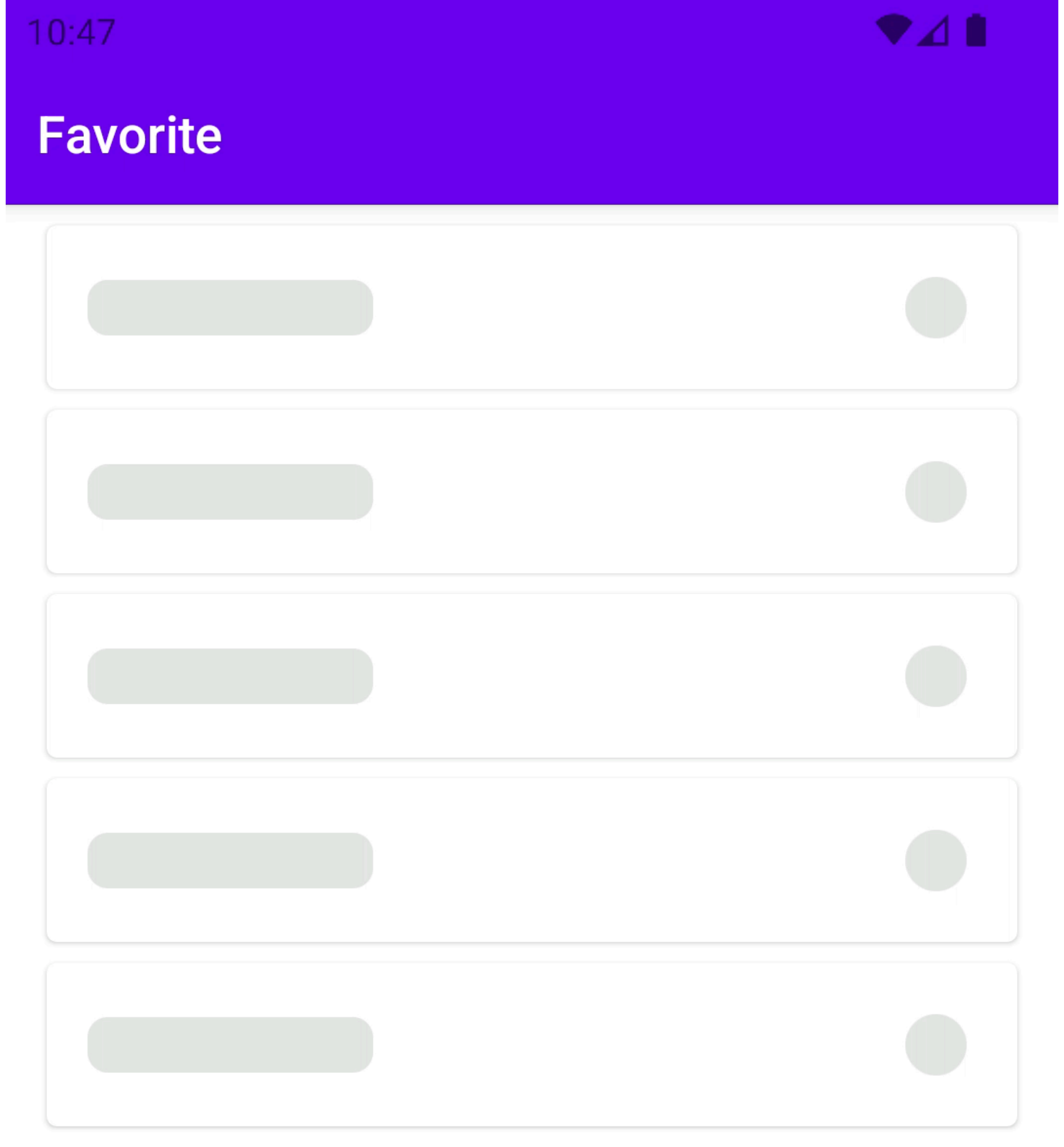
1. State - list of favorite
2. Effects - showing snackbars
3. Msg - item and favorite click

UDF sample: favorite list

1. **State - list of favorite**
2. Effects - showing snackbars
3. Msg - item and favorite click

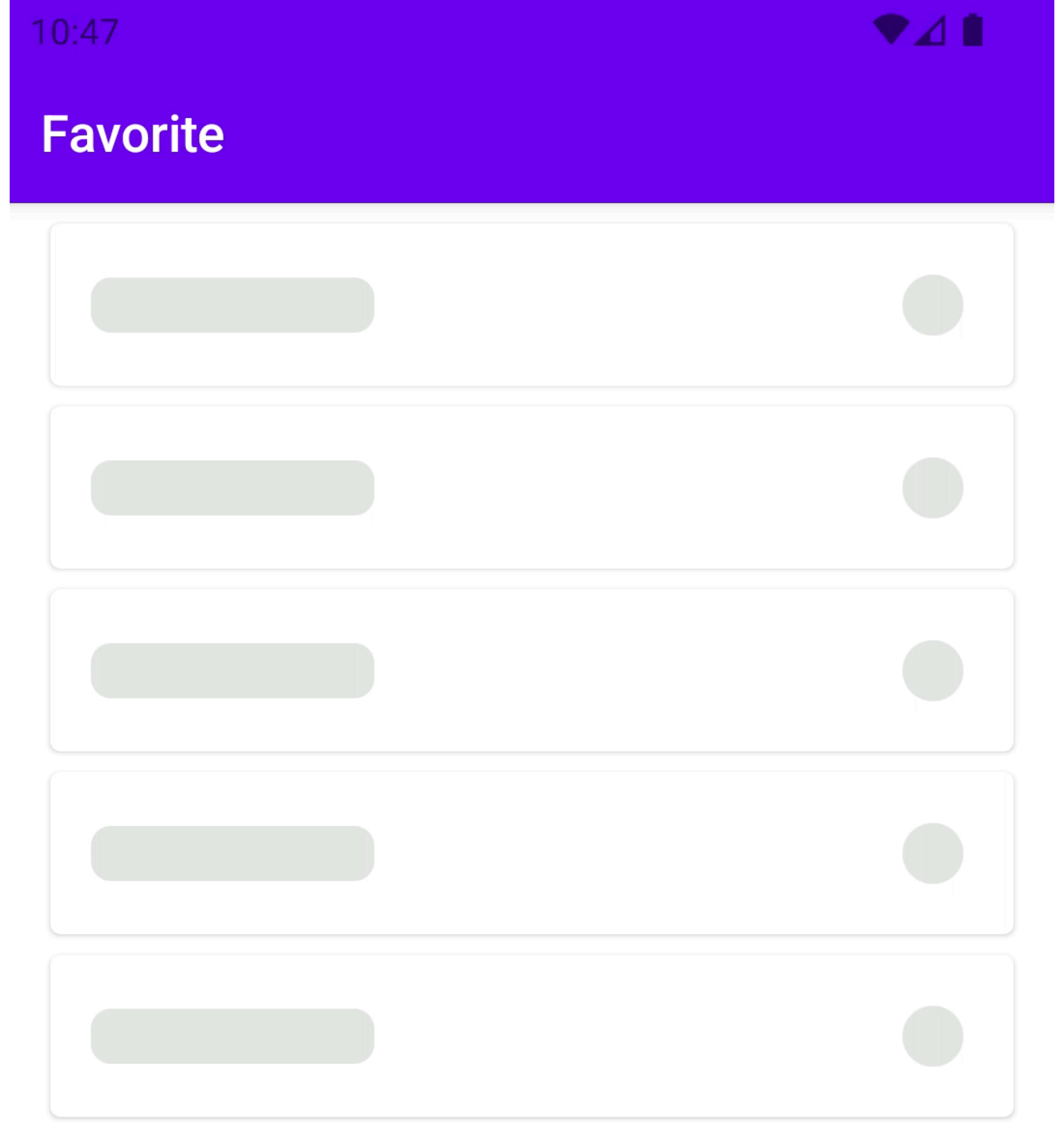


UDF sample: favorite list



1. State - list of favorite
2. Effects - showing snackbars
3. Msg - item and favorite click

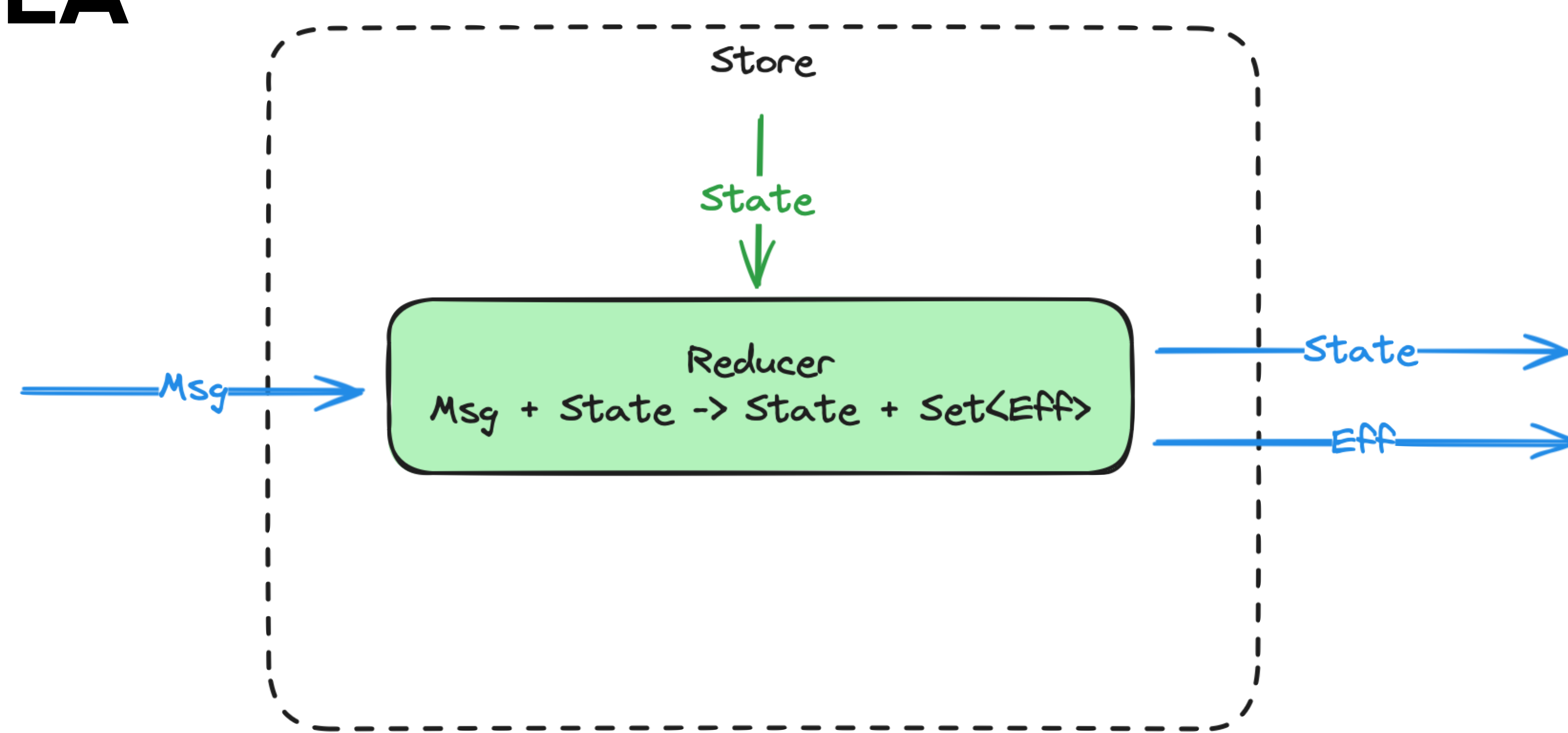
UDF sample: favorite list



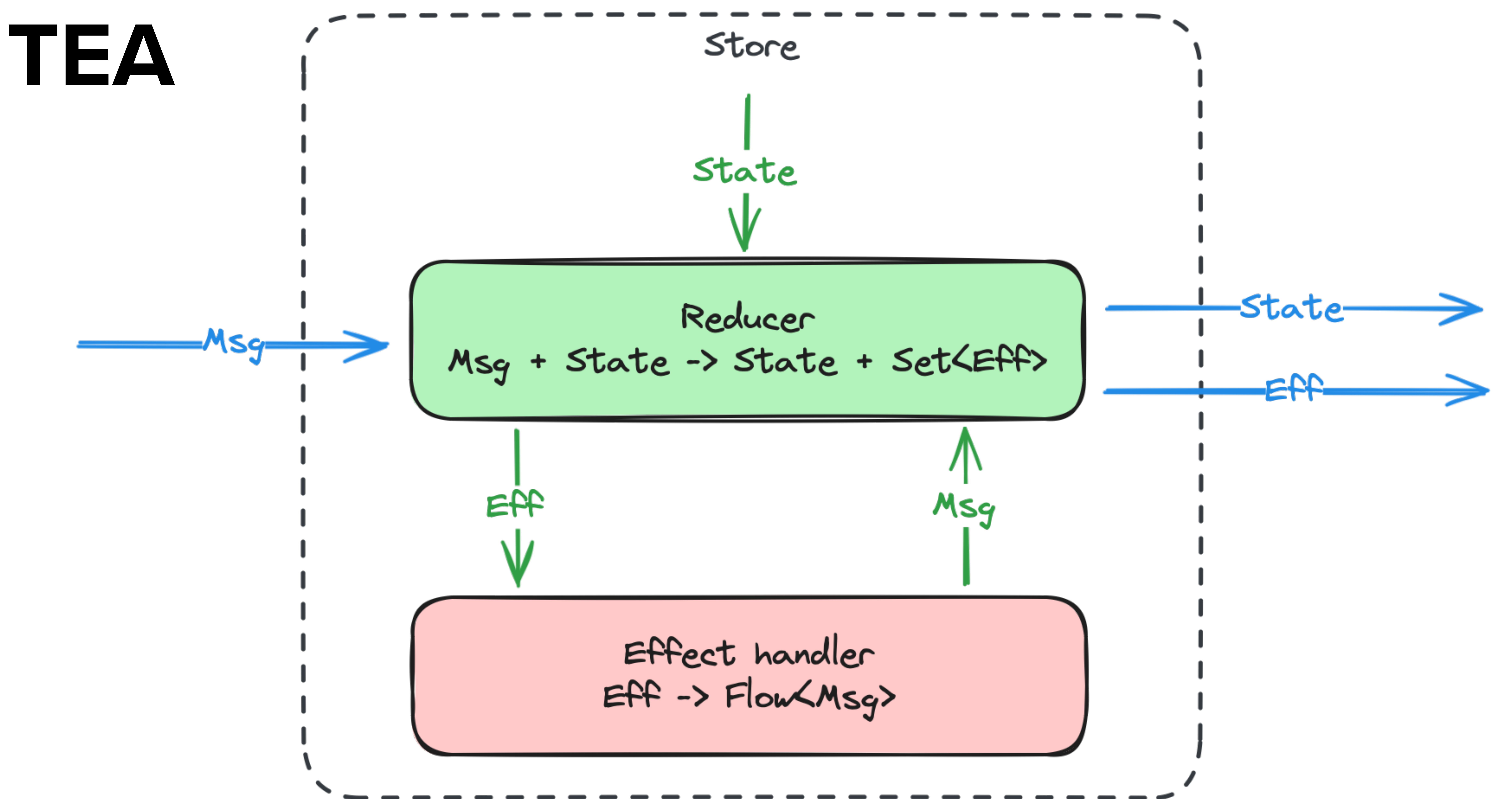
1. State - list of favorite
2. Effects - showing snackbars
3. Msg - item and favorite click

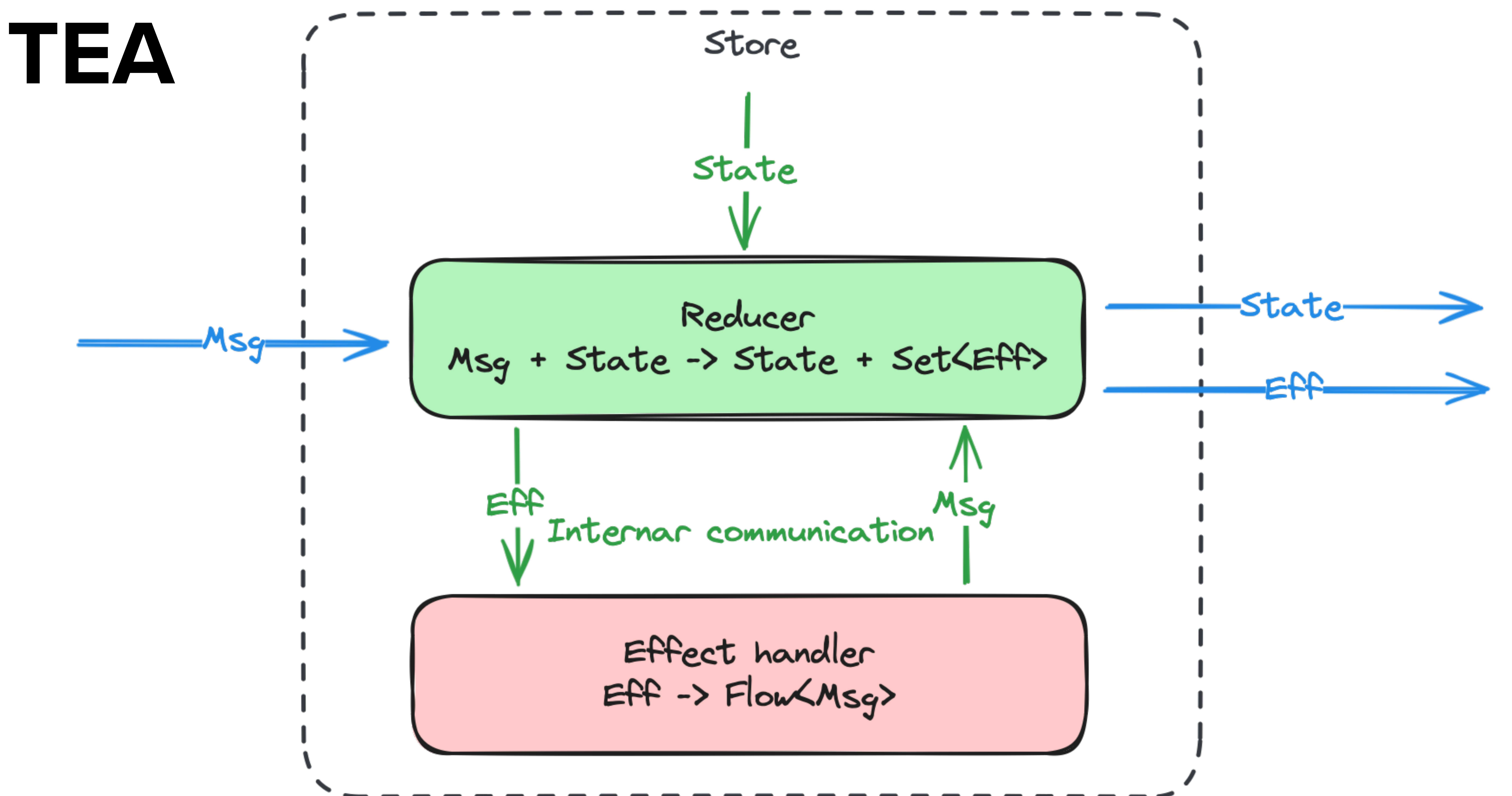
TEA (The Elm Architecture)

TEA

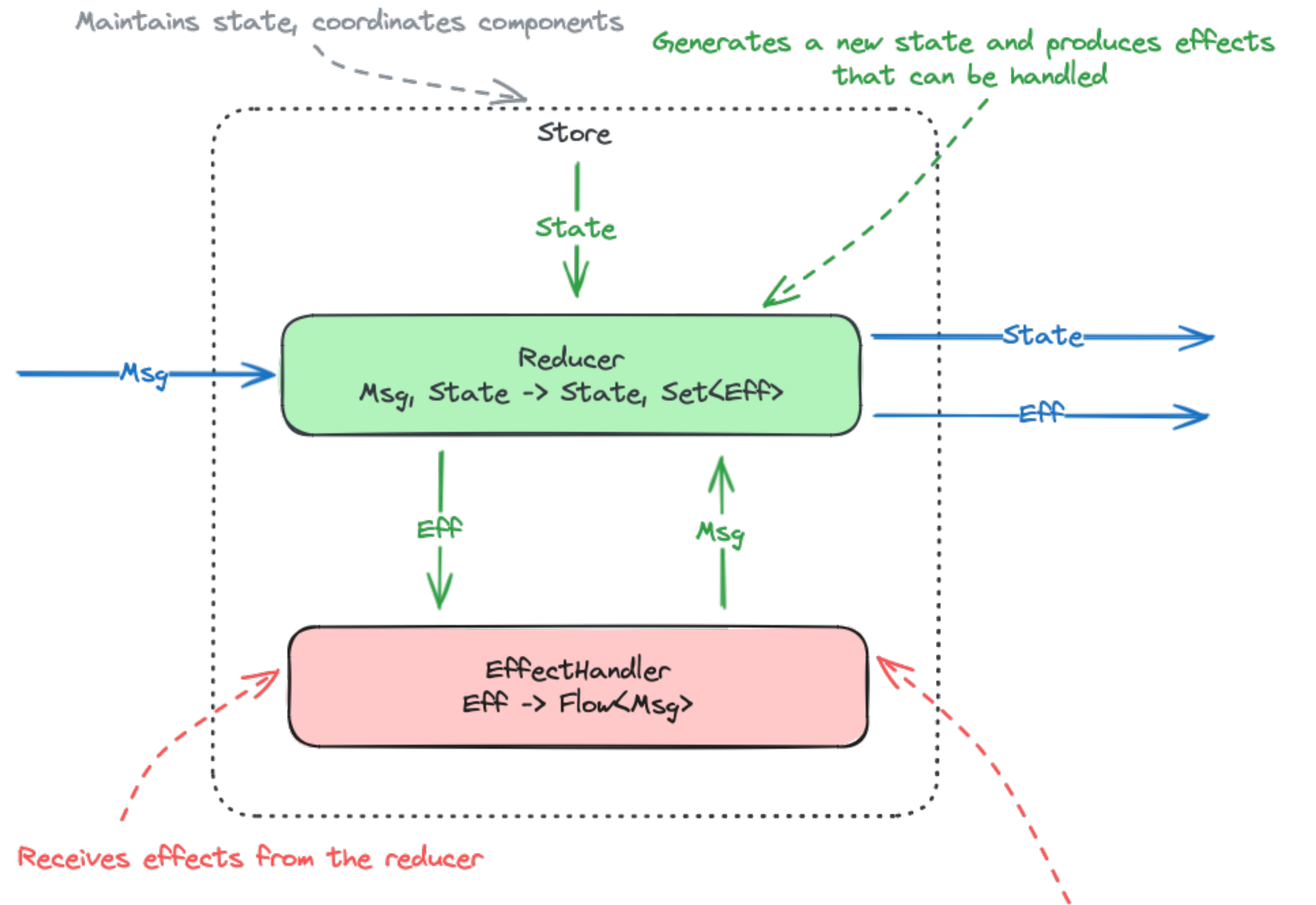


TEA





TEA
Store
State
Msg
Reducer
Msg + State -> State + Set<Eff>
State
Eff
Eff
Internar communication
Msg
Effect handler
Eff -> Flow<Msg>


Maintains state, coordinates components
Generates a new state and produces effects that can be handled
Store
State
Msg
Reducer
Msg, State -> State, Set<Eff>
State
Eff
Eff
Msg
EffectHandler
Eff -> Flow<Msg>
Receives effects from the reducer
Has the ability to send messages back to trigger the reducer

TEA advantages vs Other UDF

01

Reducer -
Pure fun

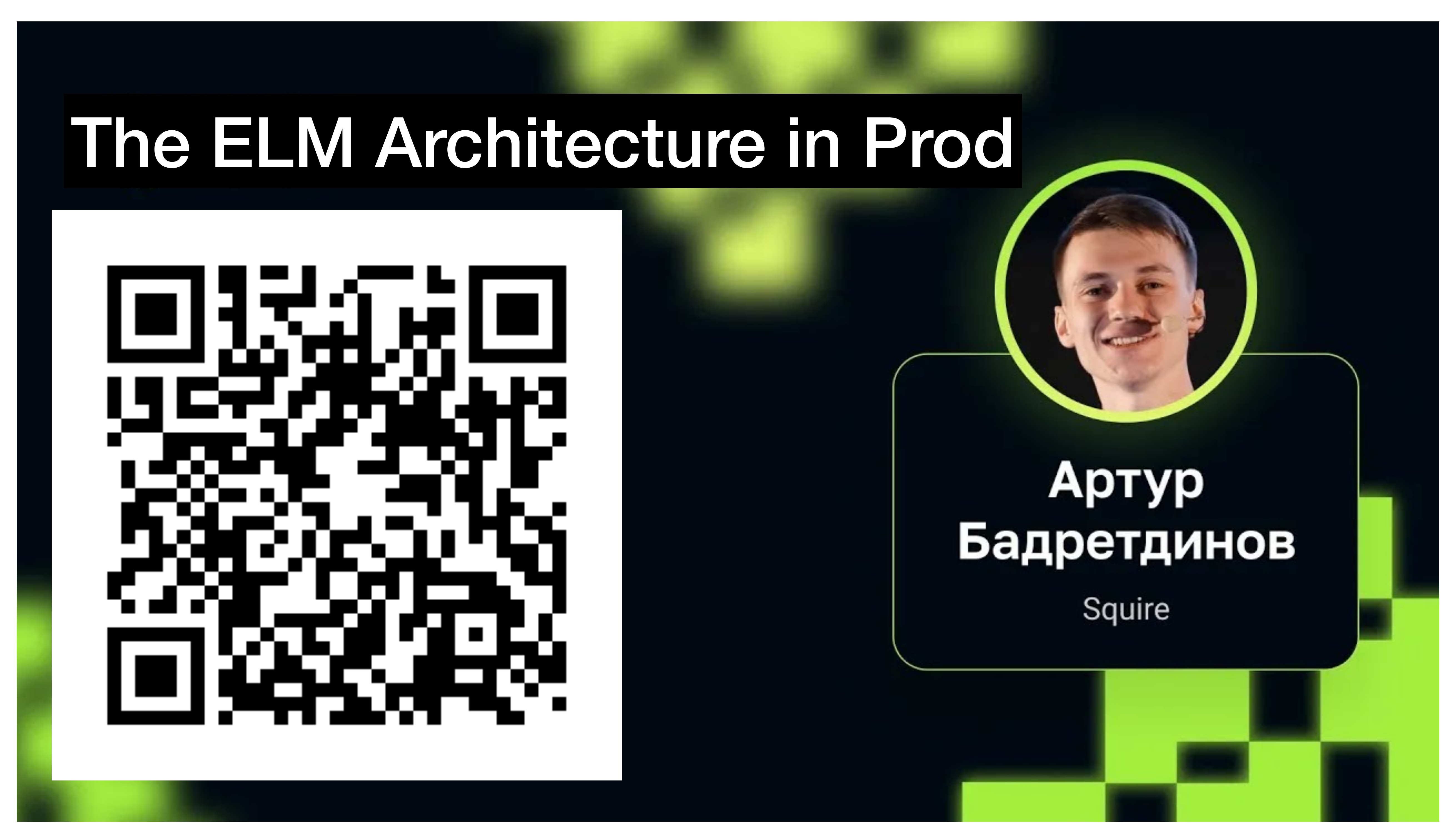
02

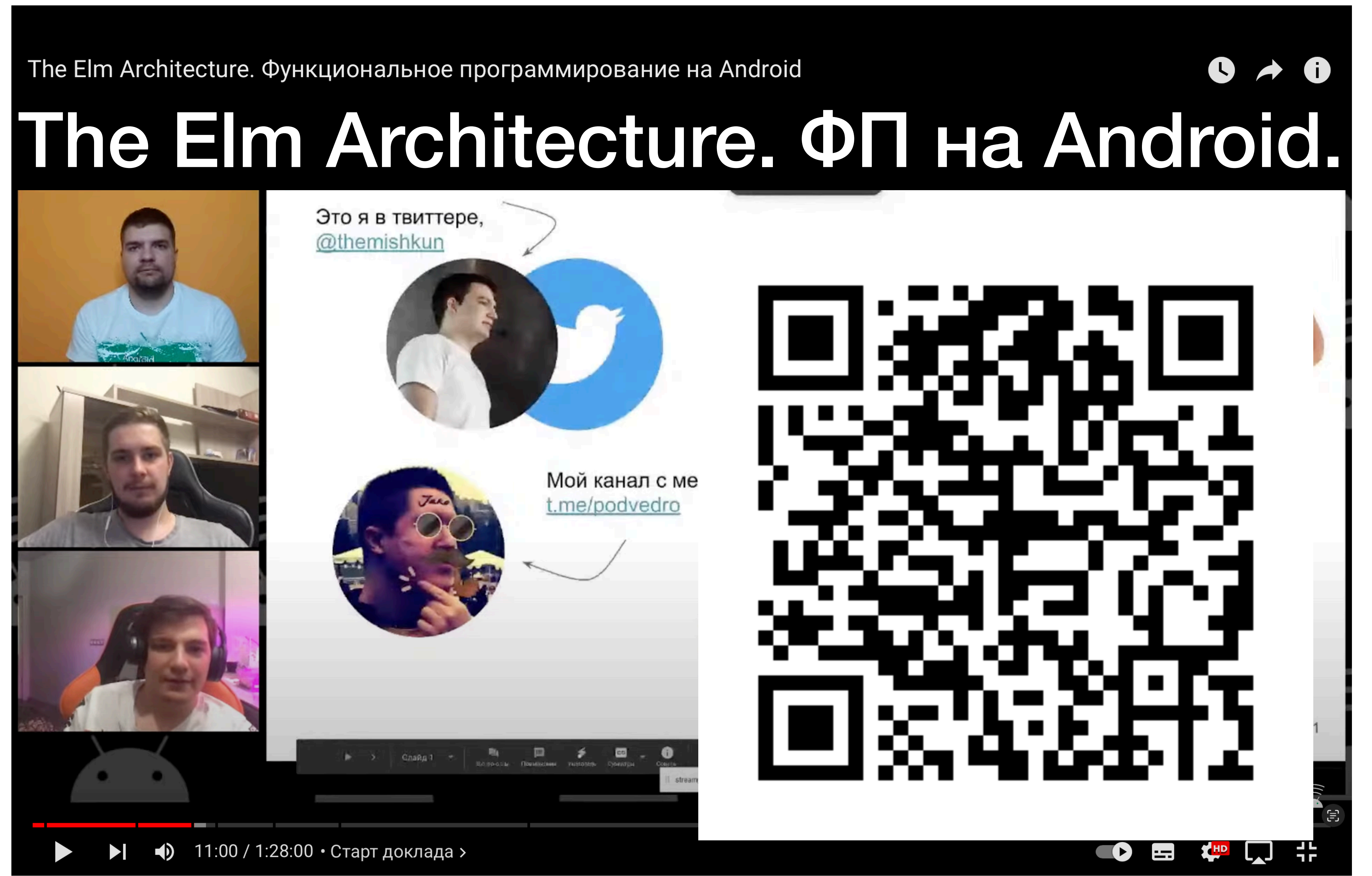
EffectHandler -
Dirty fun

03

Only reducer
accesses the State

TEA Speeches

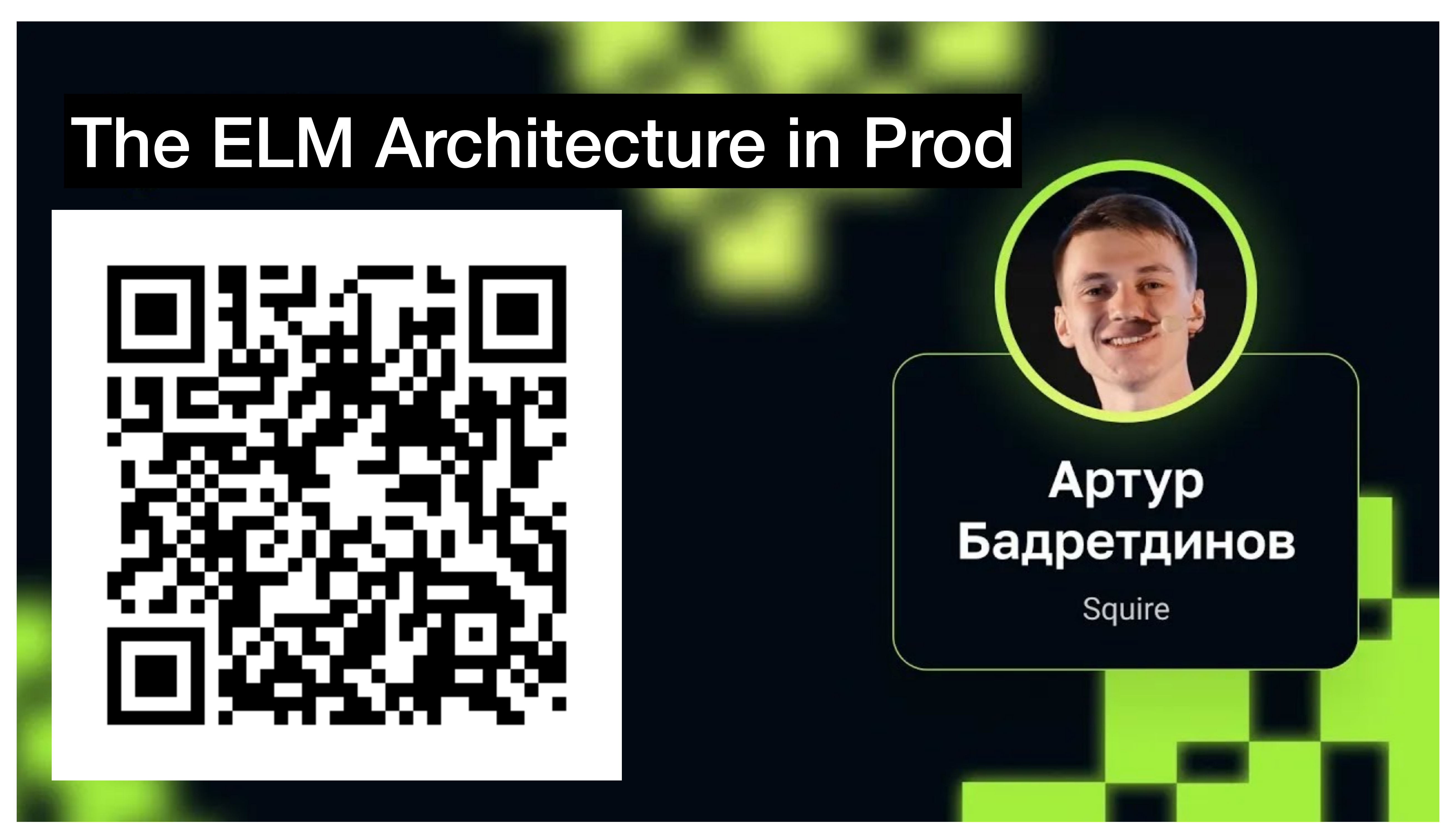


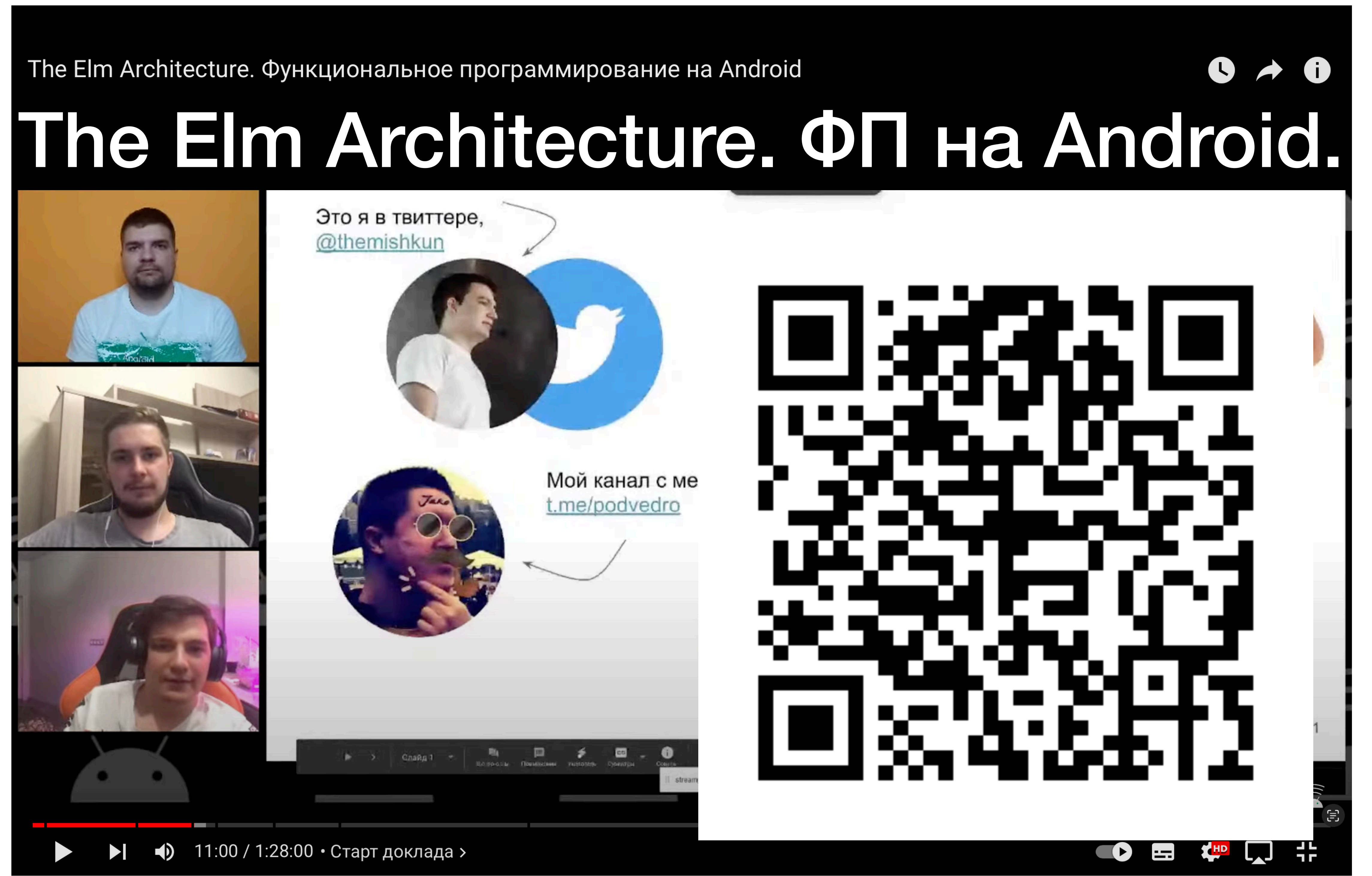


https://www.youtube.com/watch?v=ykW4UFGXAIw

https://www.youtube.com/live/5DWuNTVFaXM?si=ghXoHiYWhOEeT_IF

TEA Speeches





https://www.youtube.com/watch?v=ykW4UFGXAIw

https://www.youtube.com/live/5DWuNTVFaXM?si=ghXoHiYWhOEeT_lF

OK, but with with Real life?

Real-life questions

1. **What should be in store?**
2. How to reuse logic? F.e. pagination
3. How to incapsulate some logic?
4. How to deal with existed code?
5. How to build feature to feature communication?

Real-life questions

1. What should be in store?
2. How to reuse logic? F.e. pagination
3. How to incapsulate some logic?
4. How to deal with existed code?
5. How to build feature to feature communication?

Real-life questions

1. What should be in store?
2. How to reuse logic? F.e. pagination
3. How to incapsulate some logic?
4. How to deal with existed code?
5. How to build feature to feature communication?

Real-life questions

1. What should be in store?
2. How to reuse logic? F.e. pagination
3. How to incapsulate some logic?
4. How to deal with existed code?
5. How to build feature to feature communication?

Real-life questions

1. What should be in store?
2. How to reuse logic? F.e. pagination
3. How to incapsulate some logic?
4. How to deal with existed code?
5. How to build feature to feature communication?

Agenda

1. Build favorite feature and make it grow
2. Build pagination upon in it
3. Features integration and communication
4. Lifecycle, Long-running tasks and etc
5. How to migrate to UDF arch

Library + Sample code

https://github.com/ikarenkov/Kombucha-UDF

Agenda

1. **Build favorite feature and make it grow**
2. Build pagination upon in it
3. Features integration and communication
4. Lifecycle, Long-running tasks and etc
5. How to migrate to UDF arch

Library + Sample code

https://github.com/ikarenkov/Kombucha-UDF

Agenda

1. **Build favorite feature and make it grow**
2. **Build pagination upon in it**
3. Features integration and communication
4. Lifecycle, Long-running tasks and etc
5. How to migrate to UDF arch

Library + Sample code

https://github.com/ikarenkov/Kombucha-UDF

Agenda

1. Build favorite feature and make it grow
2. Build pagination upon in it
3. Features integration and communication
4. Lifecycle, Long-running tasks and etc
5. How to migrate to UDF arch

Library + Sample code

https://github.com/ikarenkov/Kombucha-UDF

Agenda

1. Build favorite feature and make it grow
2. Build pagination upon in it
3. Features integration and communication
4. Lifecycle, Long-running tasks and etc
5. How to migrate to UDF arch

Library + Sample code

https://github.com/ikarenkov/Kombucha-UDF

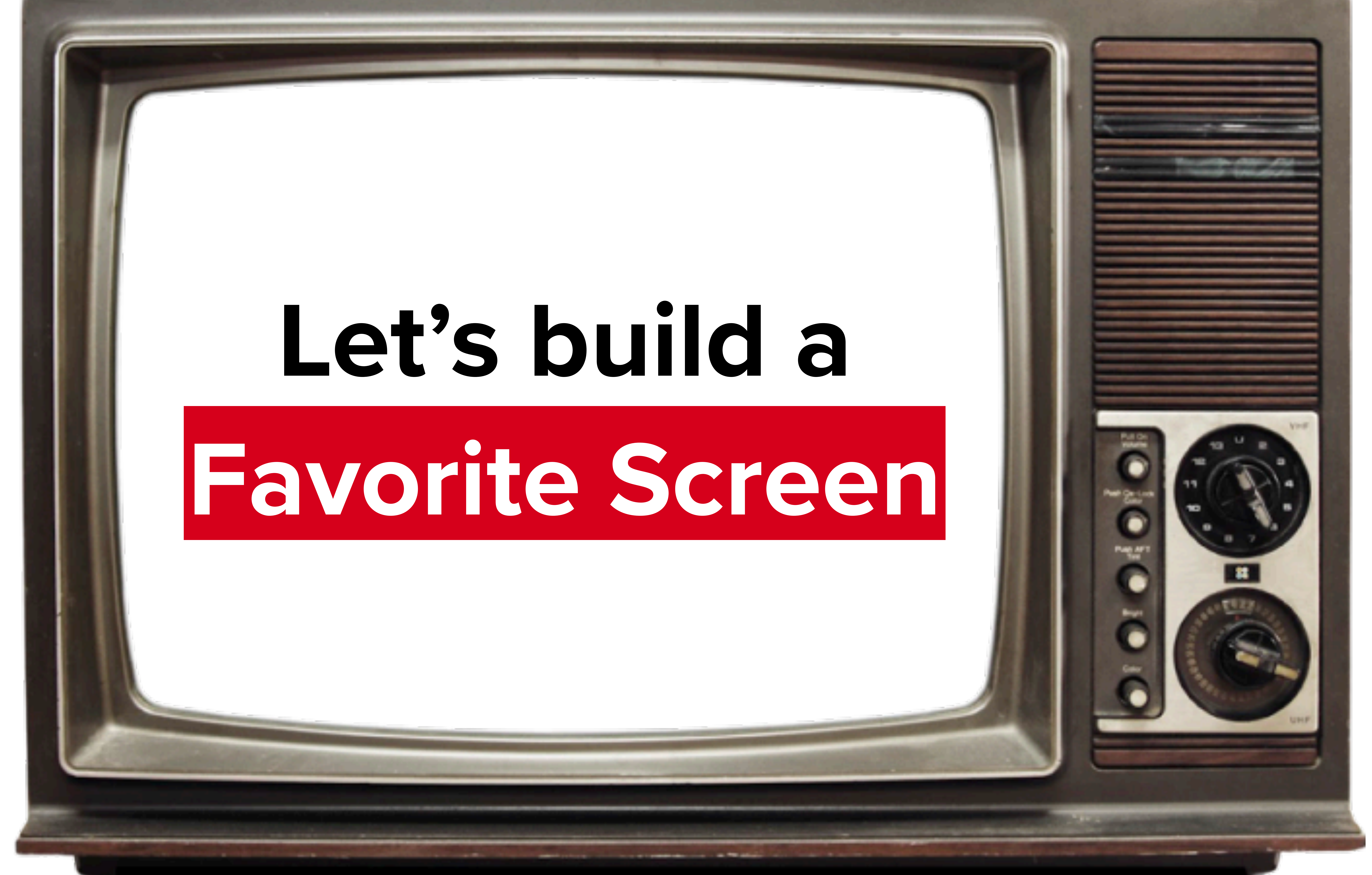
Agenda

1. Build favorite feature and make it grow
2. Build pagination upon in it
3. Features integration and communication
4. Lifecycle, Long-running tasks and etc
5. How to migrate to UDF arch

Library + Sample code

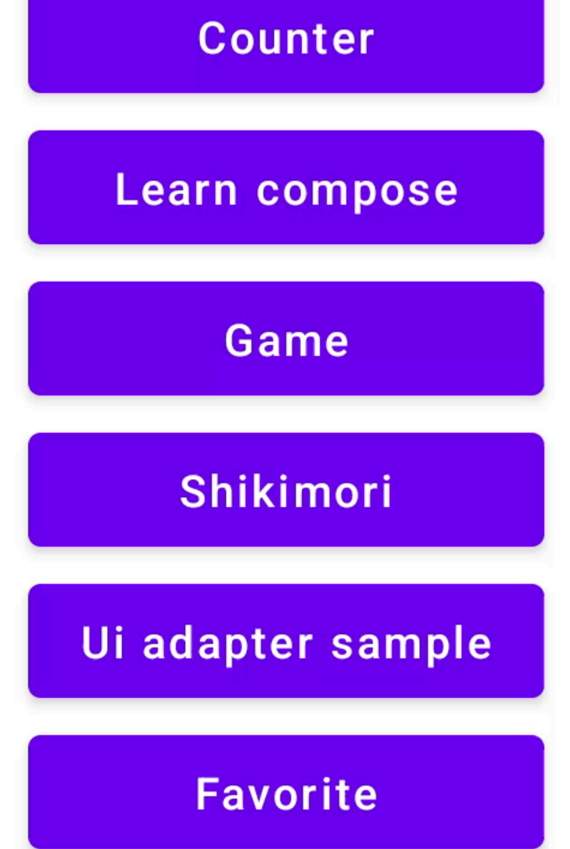
https://github.com/ikarenkov/Kombucha-UDF

sample/features/favorite/src/main/kotlin/io/github/ikarenkov/sample/favorite


Let’s build a
Favorite Screen

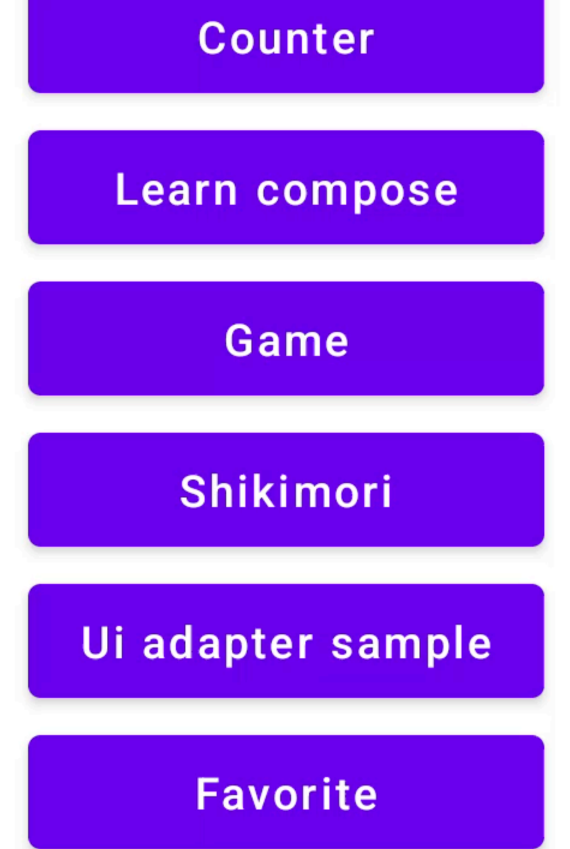
Favorite screen: features

1. **List of items**
2. Item click handling
3. Remove from favorite
4. Observe favorite updates
5. Error handling



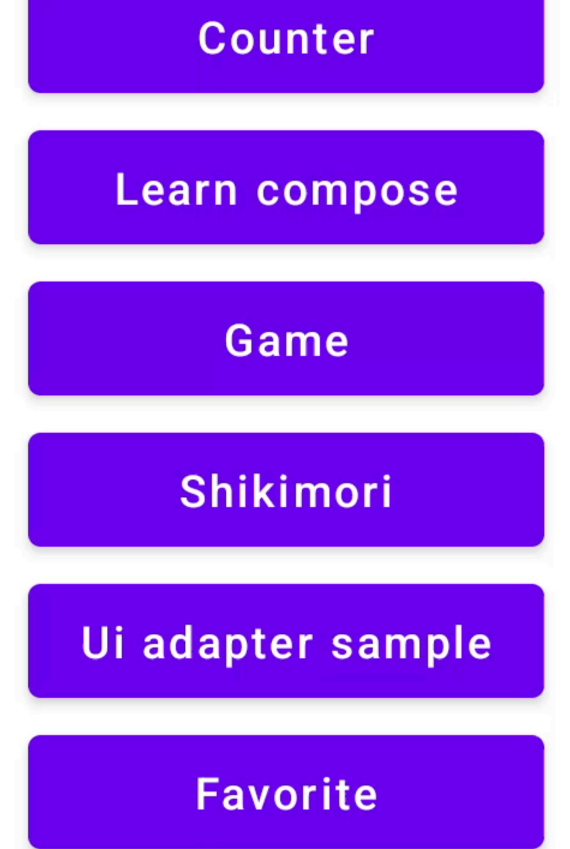
Favorite screen: features

1. List of items
2. Item click handling
3. Remove from favorite
4. Observe favorite updates
5. Error handling



Favorite screen: features

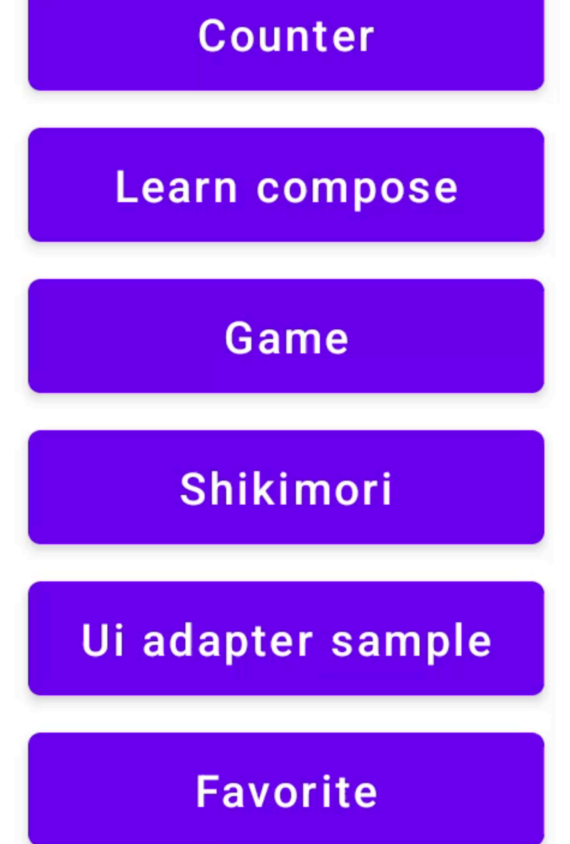
1. List of items
2. Item click handling
3. Remove from favorite
4. Observe favorite updates
5. Error handling



12:57

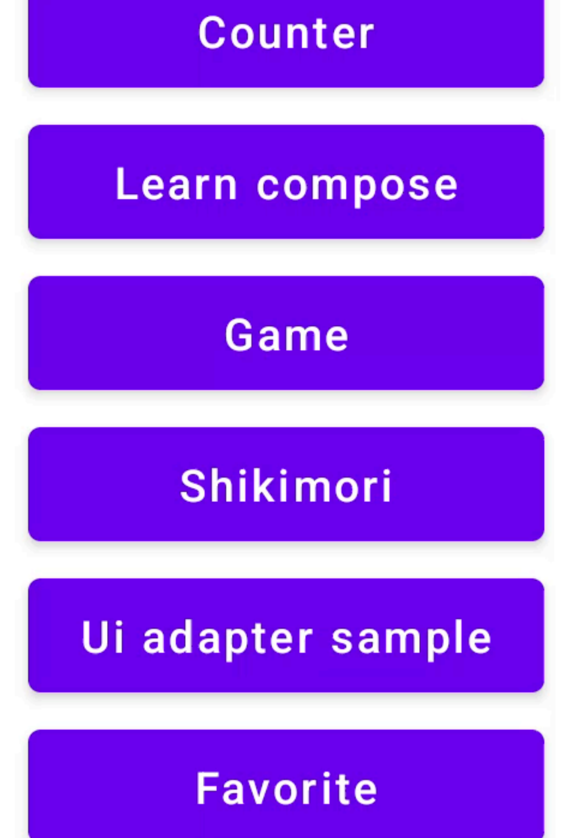
Favorite screen: features

1. List of items
2. Item click handling
3. Remove from favorite
4. Observe favorite updates
5. Error handling



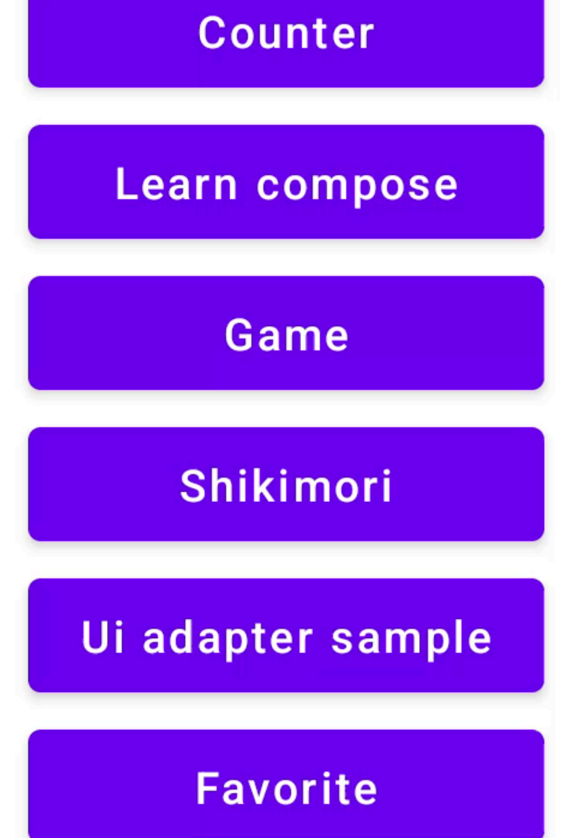
Favorite screen: features

1. List of items
2. Item click handling
3. Remove from favorite
4. Observe favorite updates
5. Error handling

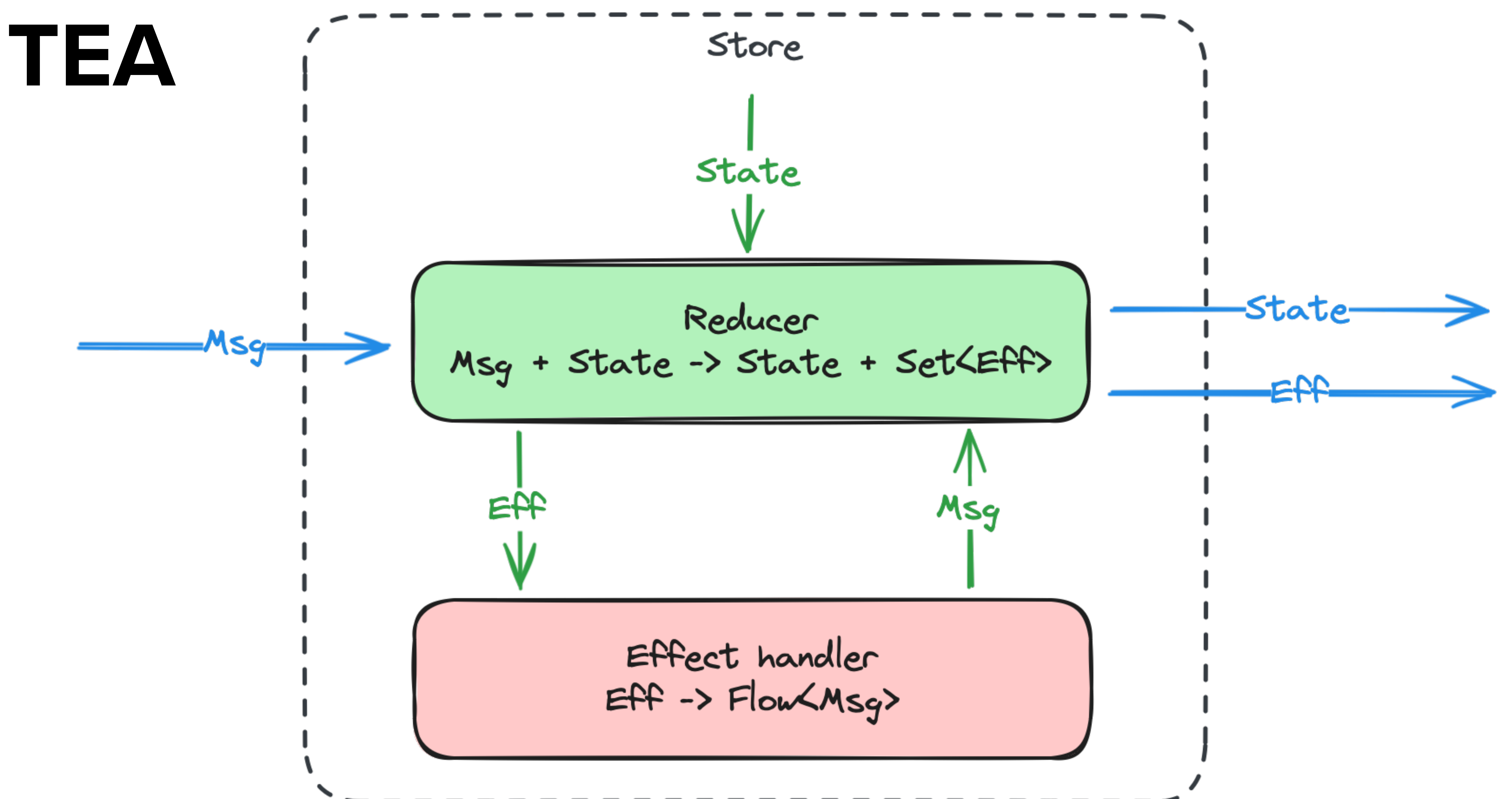


Favorite screen: features

1. List of items
2. Item click handling
3. Remove from favorite
4. Observe favorite updates
5. Error handling

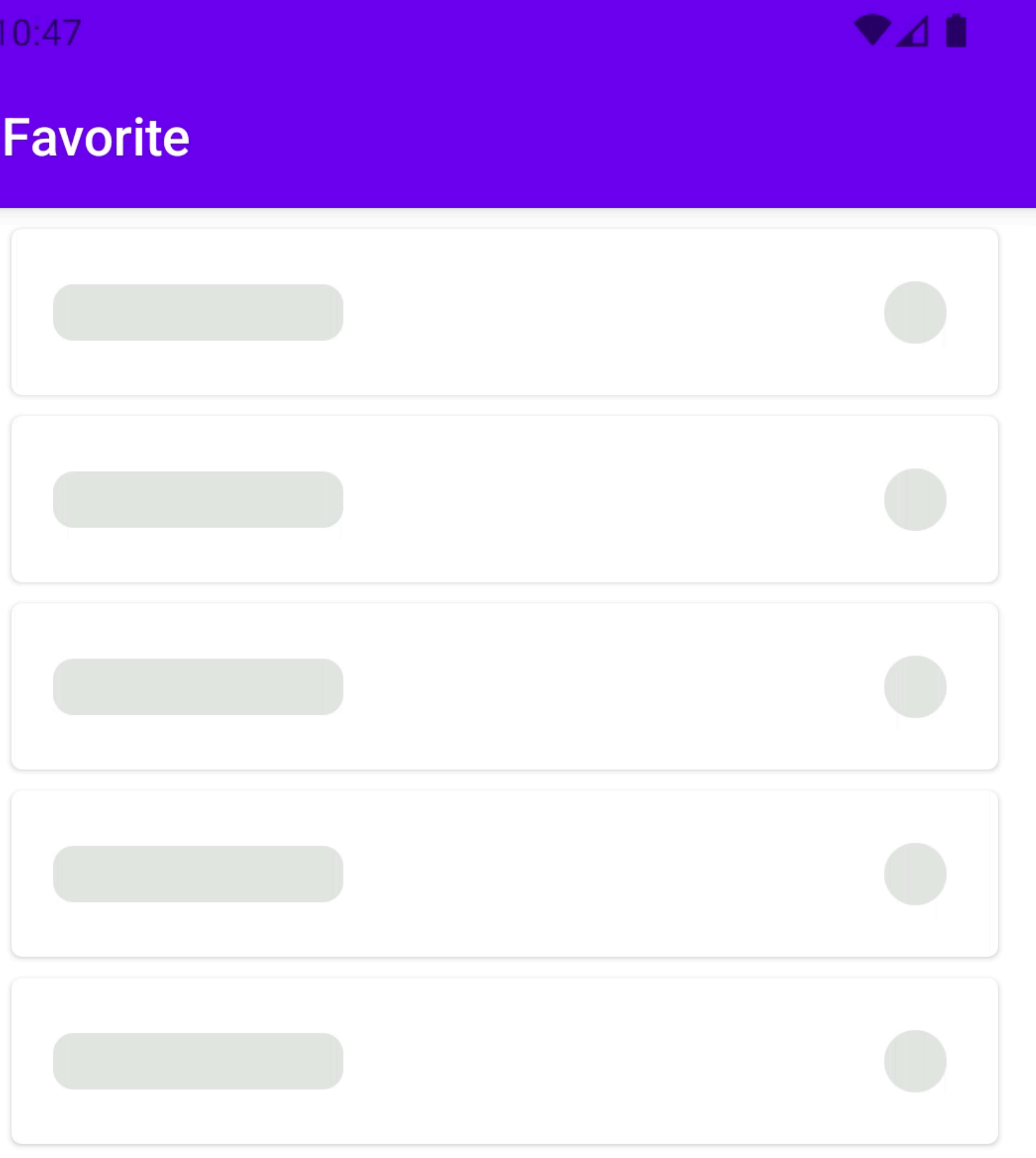


TEA



State

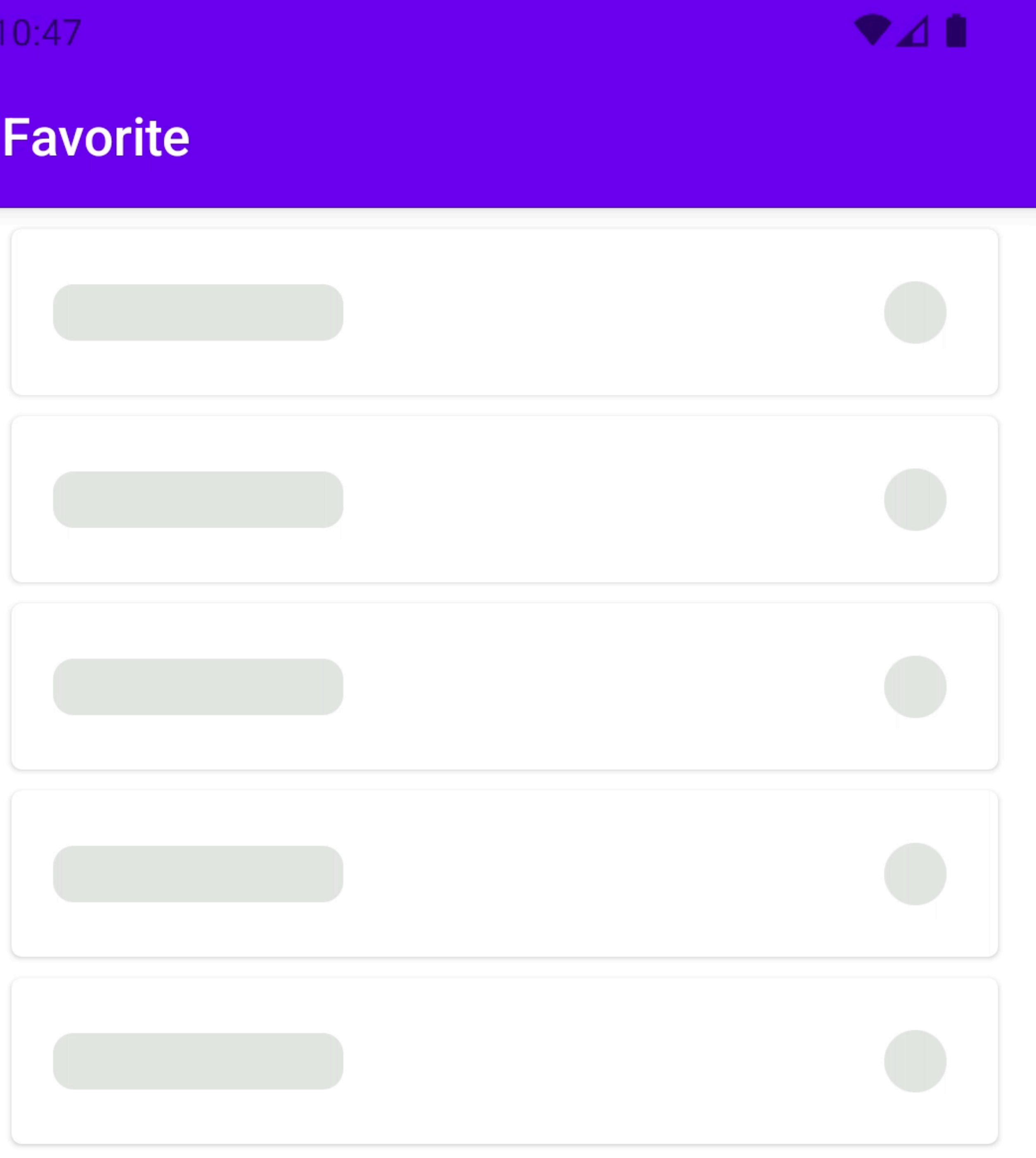
```kotlin
internal data class State(
    val content: LCE<List<FavoriteItem>>,
)

data class FavoriteItem(
    val id: String,
    val title: String
)
```



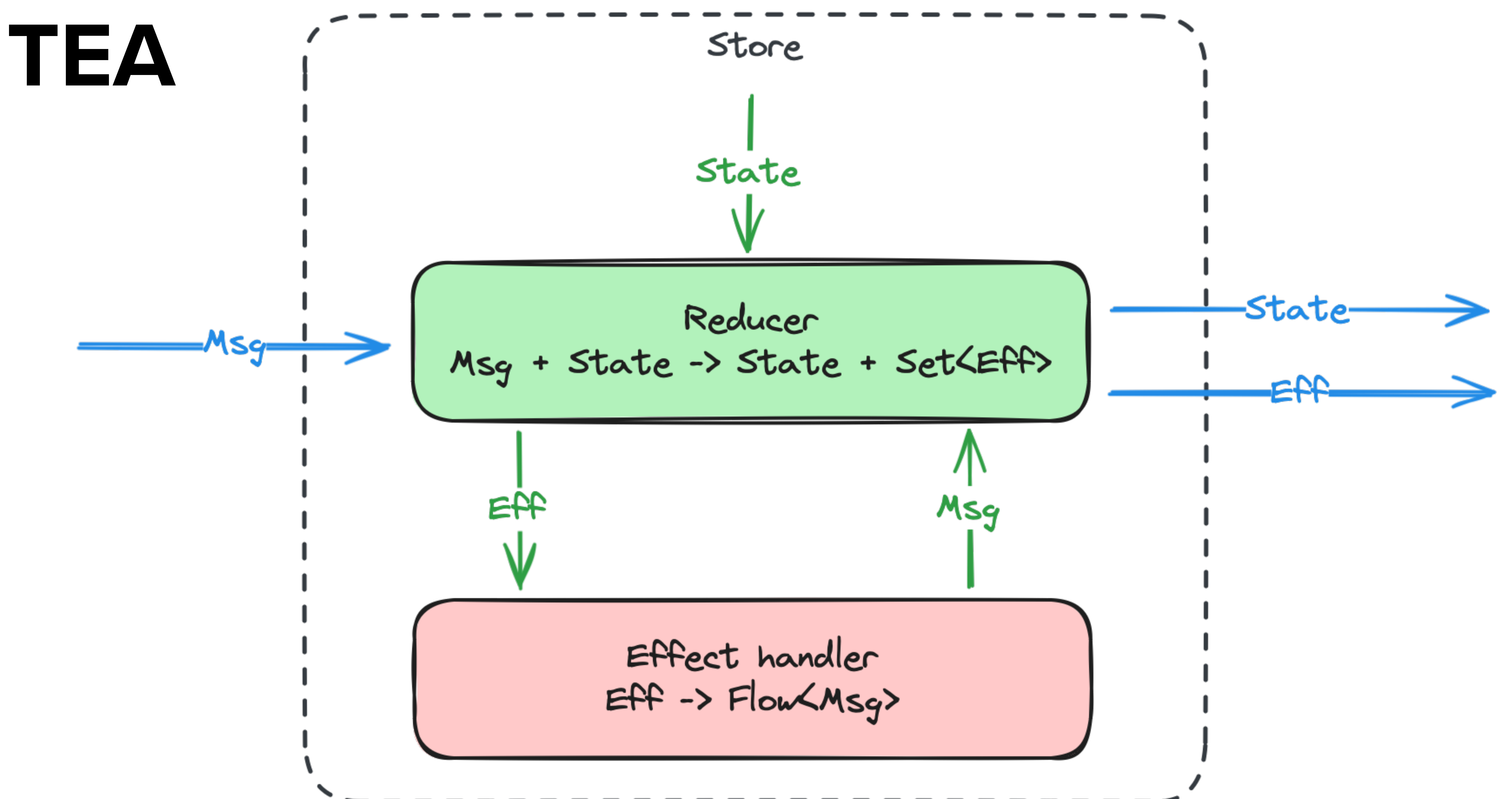
impl/FavoriteListFeature.kt

State

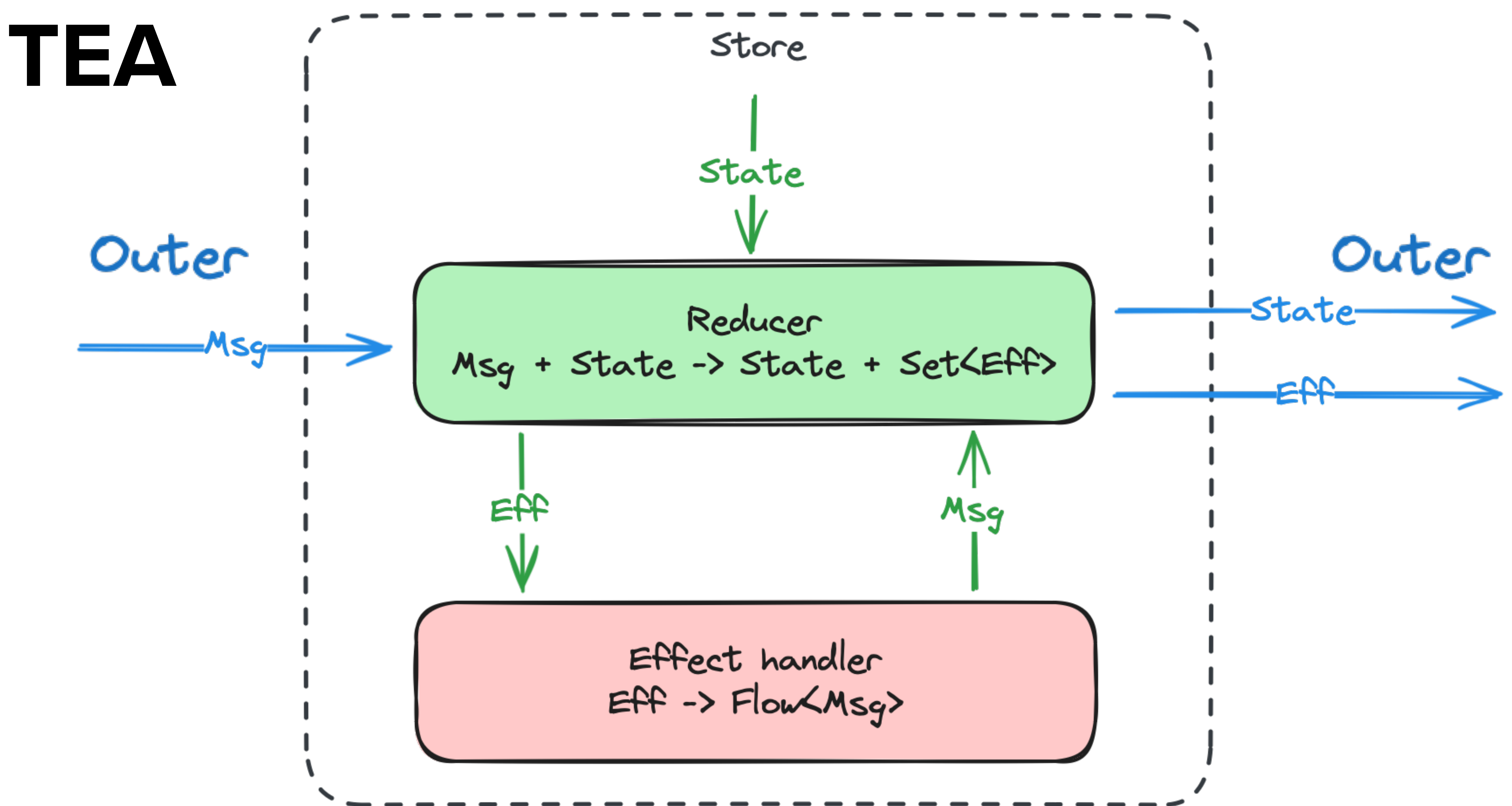
```
internal data class State(
    val content: LCE<List<FavoriteItem>>,
)

data class FavoriteItem(
    val id: String,
    val title: String
)
```



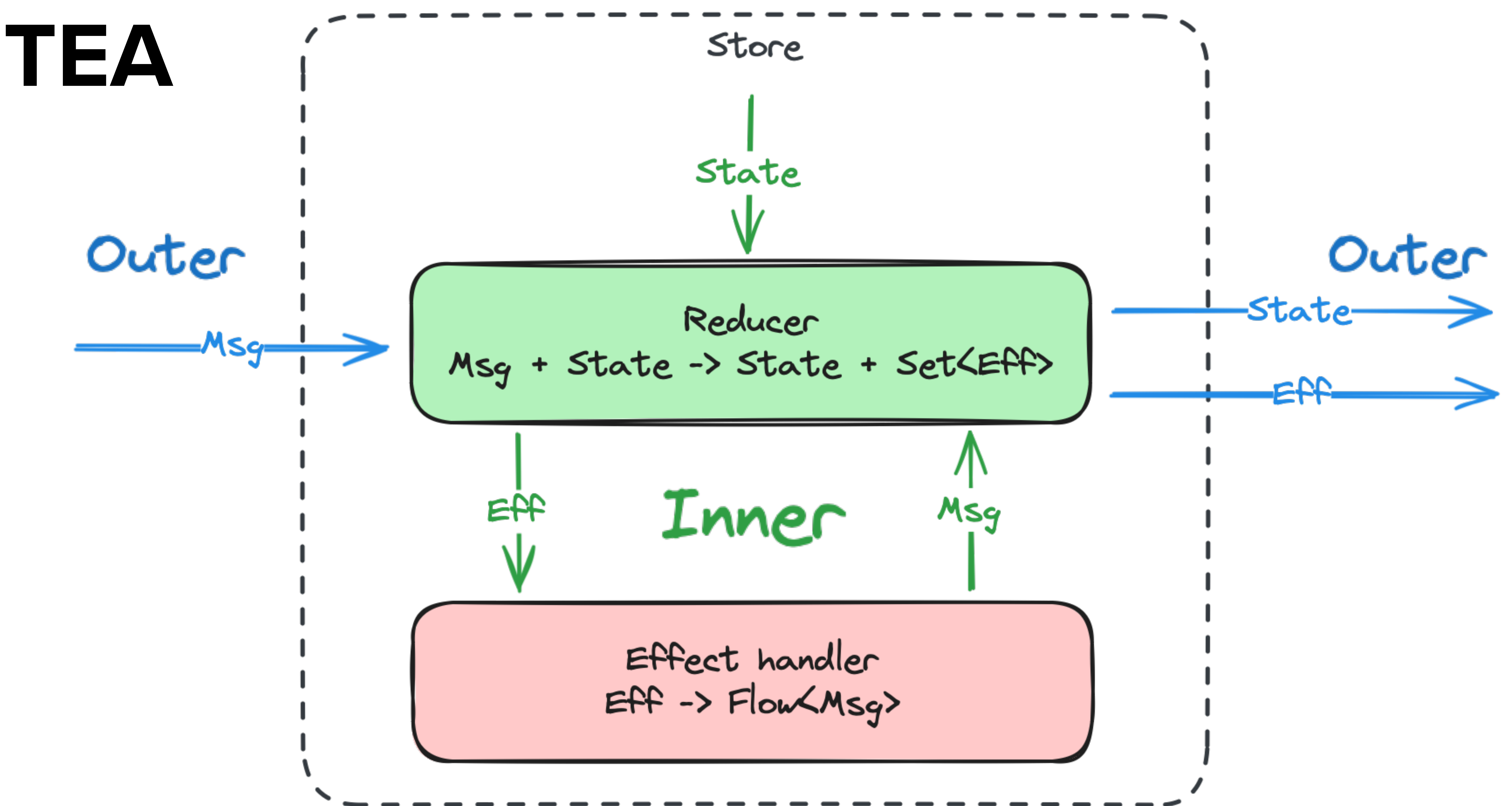
impl/FavoriteListFeature.kt


TEA
Store
State
Msg
Reducer
Msg + State -> State + Set<Eff>
State
Eff
Eff
Msg
Effect handler
Eff -> Flow<Msg>

TEA



TEA



Msg

```kotlin
sealed interface Msg {

    sealed interface Outer : Msg {
        data class ItemClick(val id: String) : Outer
        data class RemoveFavorite(val id: String) : Outer
        data object RetryLoad : Outer
    }

    sealed interface Inner : Msg {

        data class AddItem(val item: FavoriteItem) : Inner
        data class ItemLoadingResult(
            val result: Result<List<FavoriteItem>>
        ) : Inner
        sealed interface ItemRemoveResult : Inner {...}
    }
}
```

10:47

Favorite

Msg

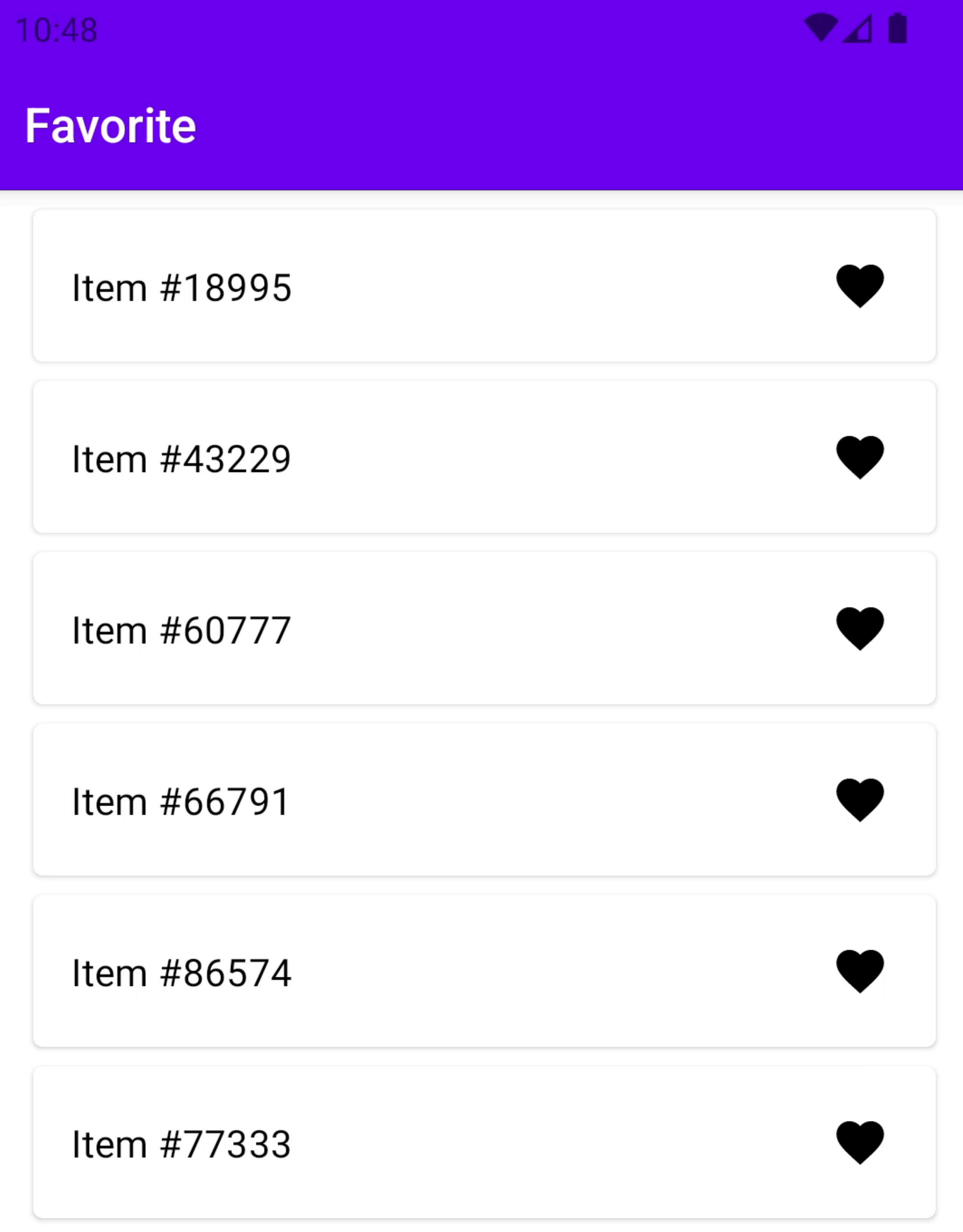
```
sealed interface Msg {

    sealed interface Outer : Msg {
        data class ItemClick(val id: String) : Outer
        data class RemoveFavorite(val id: String) : Outer
        data object RetryLoad : Outer
    }

    sealed interface Inner : Msg {

        data class AddItem(val item: FavoriteItem) : Inner
        data class ItemLoadingResult(
            val result: Result<List<FavoriteItem>>
        ) : Inner
        sealed interface ItemRemoveResult : Inner {...}
    }
}
```

10:47

Favorite

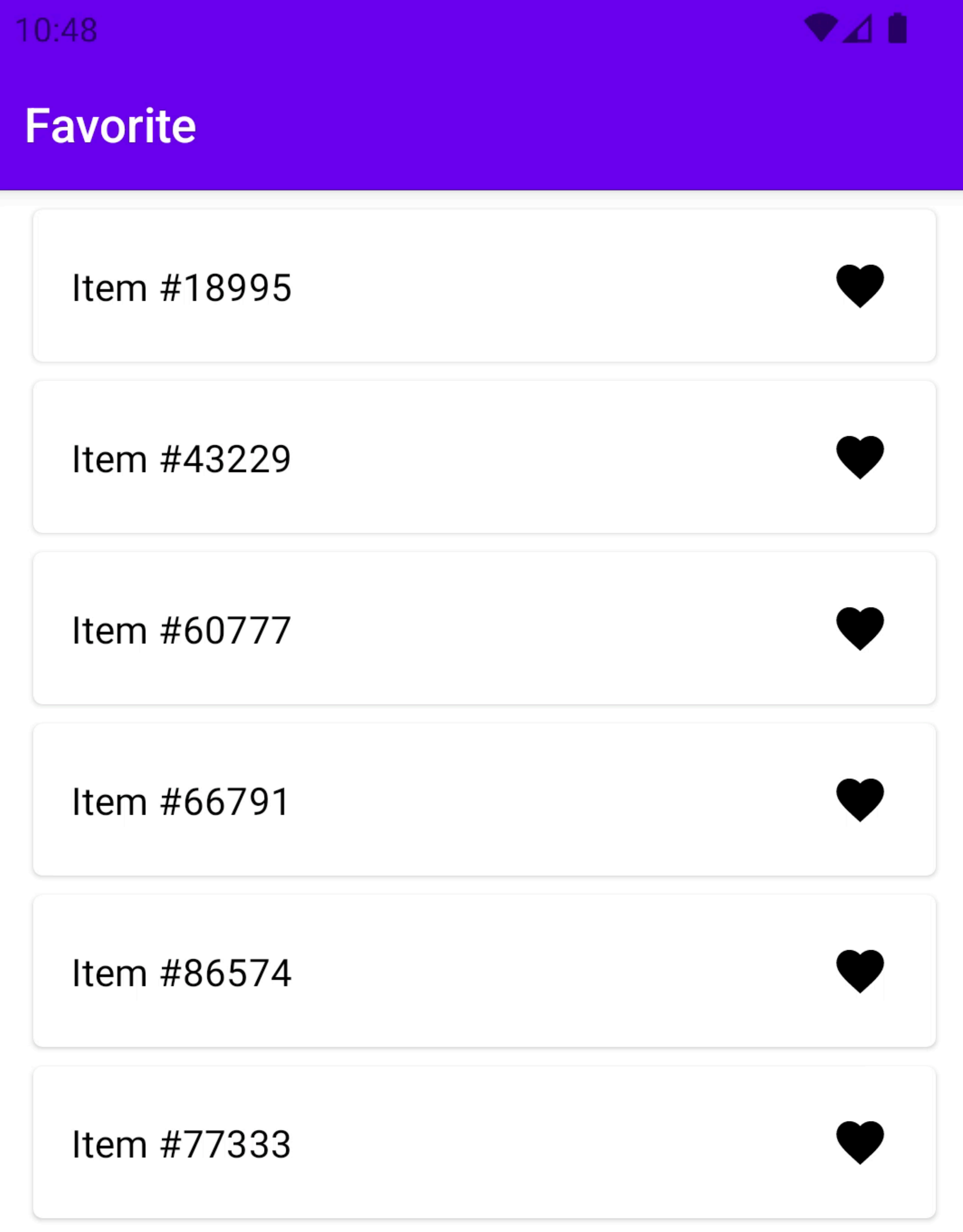
Msg

```kotlin
sealed interface Msg {

    sealed interface Outer : Msg {
        data class ItemClick(val id: String) : Outer
        data class RemoveFavorite(val id: String) : Outer
        data object RetryLoad : Outer
    }

    sealed interface Inner : Msg {

        data class AddItem(val item: FavoriteItem) : Inner
        data class ItemLoadingResult(
            val result: Result<List<FavoriteItem>>
        ) : Inner
        sealed interface ItemRemoveResult : Inner {...}
    }
}
```



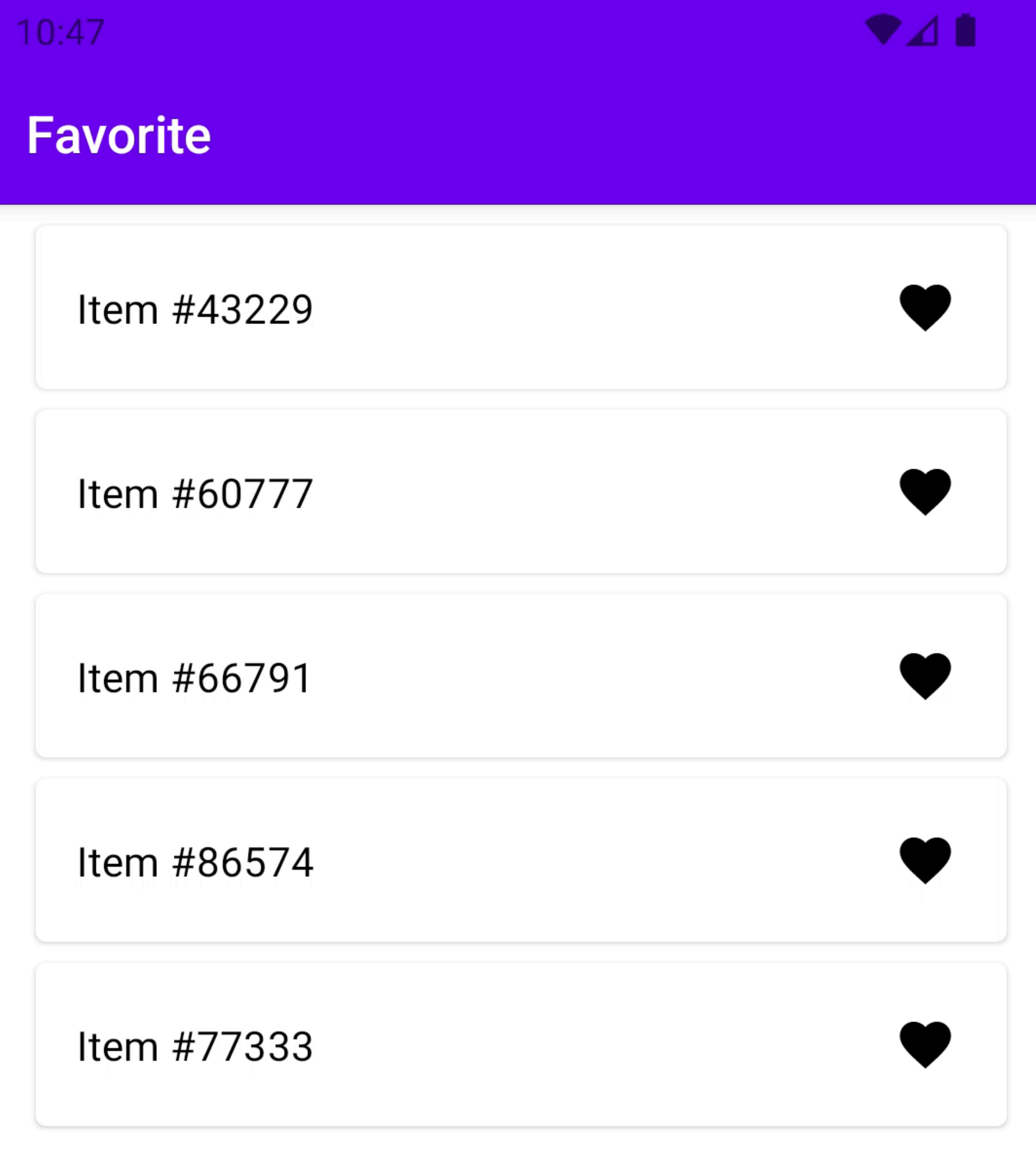
Msg

```kotlin
sealed interface Msg {

    sealed interface Outer : Msg {
        data class ItemClick(val id: String) : Outer
        data class RemoveFavorite(val id: String) : Outer
        data object RetryLoad : Outer
    }

    sealed interface Inner : Msg {

        data class AddItem(val item: FavoriteItem) : Inner
        data class ItemLoadingResult(
            val result: Result<List<FavoriteItem>>
        ) : Inner
        sealed interface ItemRemoveResult : Inner {...}
    }
}
```



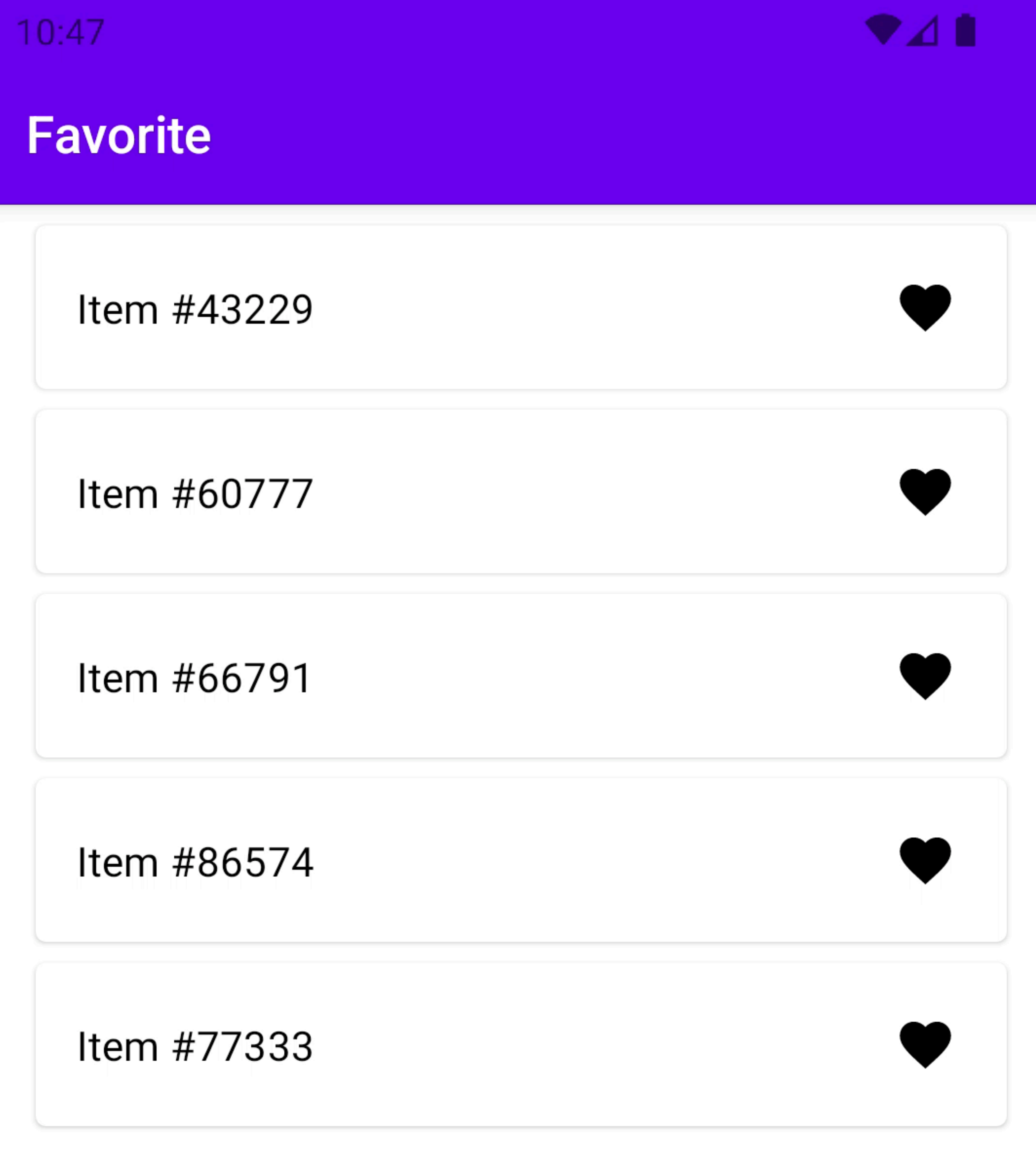
Msg

```
sealed interface Msg {

    sealed interface Outer : Msg {
        data class ItemClick(val id: String) : Outer
        data class RemoveFavorite(val id: String) : Outer
        data object RetryLoad : Outer
    }

    sealed interface Inner : Msg {

        data class AddItem(val item: FavoriteItem) : Inner
        data class ItemLoadingResult(
            val result: Result<List<FavoriteItem>>
        ) : Inner
        sealed interface ItemRemoveResult : Inner {...}
    }
}
```



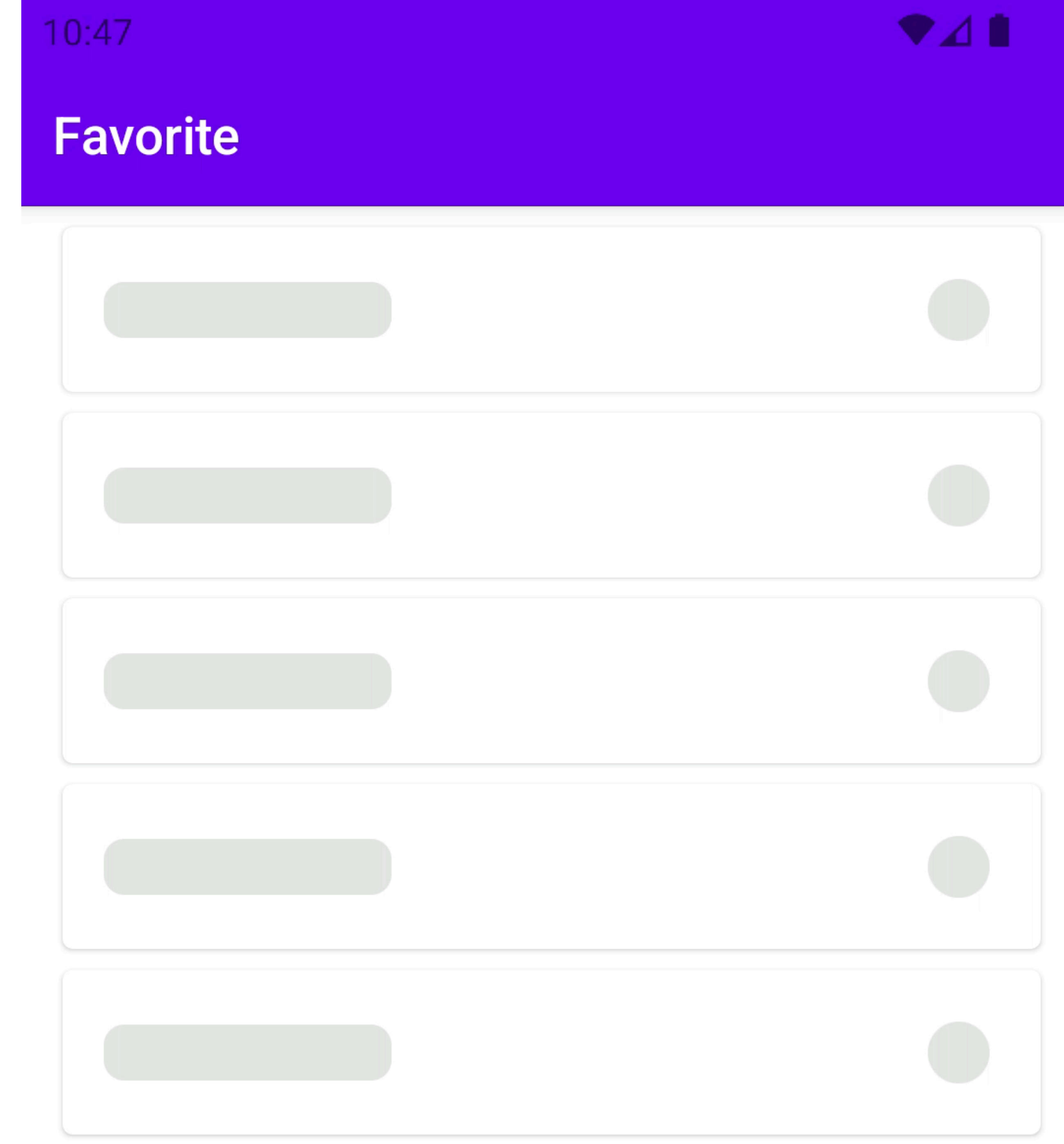
Msg

```kotlin
sealed interface Msg {

    sealed interface Outer : Msg {
        data class ItemClick(val id: String) : Outer
        data class RemoveFavorite(val id: String) : Outer
        data object RetryLoad : Outer
    }

    sealed interface Inner : Msg {

        data class AddItem(val item: FavoriteItem) : Inner
        data class ItemLoadingResult(
            val result: Result<List<FavoriteItem>>
        ) : Inner
        sealed interface ItemRemoveResult : Inner {...}
    }
}
```



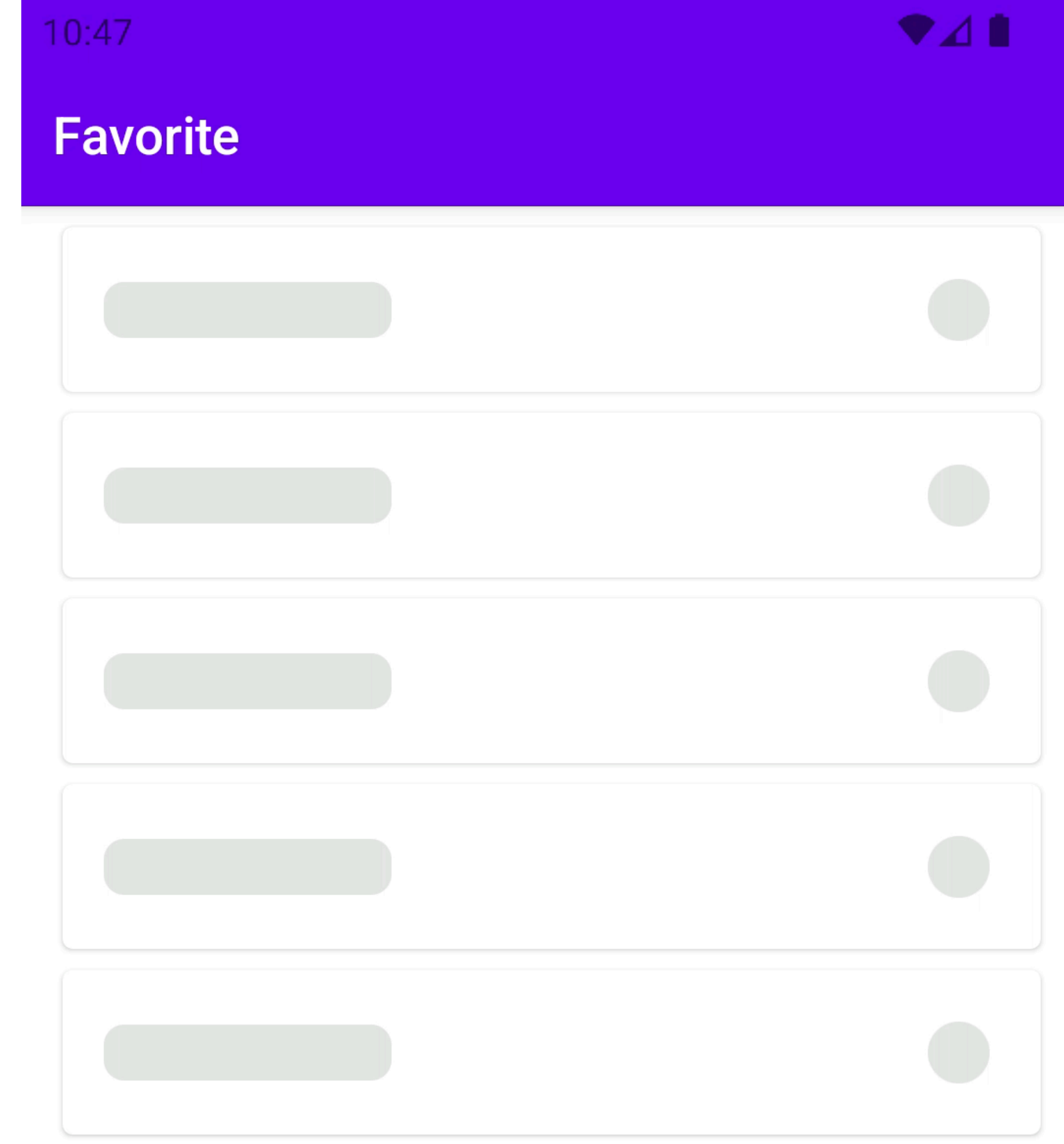
Reducer

```
val reducer = dslReducer<Msg, State, Eff> { msg ->
    when (msg) {
        is Msg.Outer -> outerReducer(msg)
        is Msg.Inner -> innerReducer(msg)
    }
}
```



Reducer

```
val reducer = dslReducer<Msg, State, Eff> { msg ->
    when (msg) {
        is Msg.Outer -> outerReducer(msg)
        is Msg.Inner -> innerReducer(msg)
    }
}
```



Reducer

```
private fun ResultBuilder<State, Eff>.outerReducer(msg: Msg.Outer) {
    when (msg) {
        is Msg.Outer.RemoveFavorite -> ...
        is Msg.Outer.RetryLoad -> {
            if (state.content is LCE.Error && !state.content.inProgress) {
                state { State(LCE.Loading()) }
                eff(Eff.Inner.LoadFav)
            }
        }
        is Msg.Outer.ItemClick -> eff(Eff.Outer.ItemClick(msg.id))
    }
}
```

Reducer

```
private fun ResultBuilder<State, Eff>.outerReducer(msg: Msg.Outer) {
    when (msg) {
        is Msg.Outer.RemoveFavorite -> ...
        is Msg.Outer.RetryLoad -> {
            if (state.content is LCE.Error && !state.content.inProgress) {
                state { State(LCE.Loading()) }
                eff(Eff.Inner.LoadFav)
            }
        }
        is Msg.Outer.ItemClick -> eff(Eff.Outer.ItemClick(msg.id))
    }
}
```

Reducer

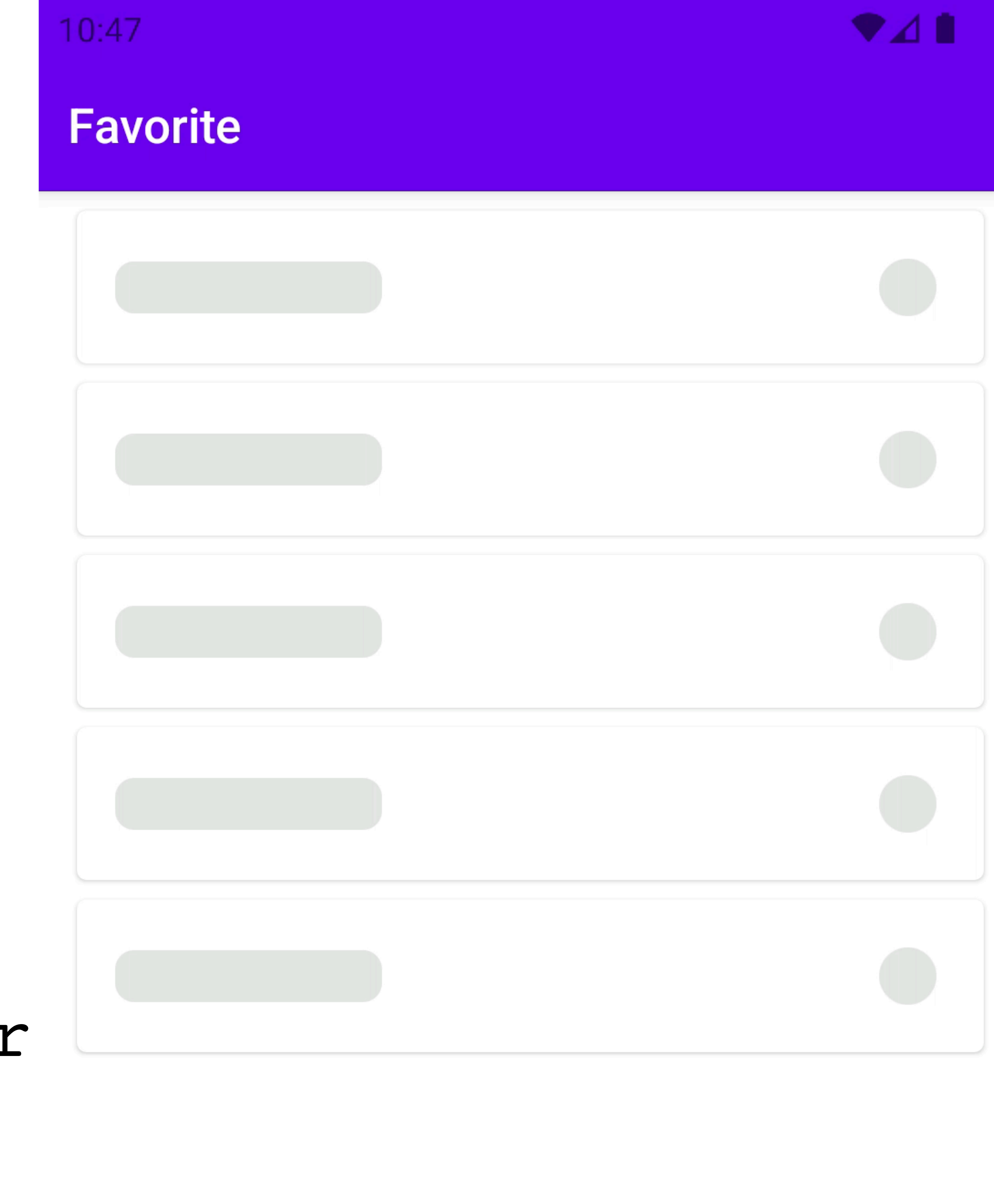
```
private fun ResultBuilder<State, Eff>.outerReducer(msg: Msg.Outer) {
    when (msg) {
        is Msg.Outer.RemoveFavorite -> ...
        is Msg.Outer.RetryLoad -> {
            if (state.content is LCE.Error && !state.content.inProgress) {
                state { State(LCE.Loading()) }
                eff(Eff.Inner.LoadFav)
            }
        }
        is Msg.Outer.ItemClick -> eff(Eff.Outer.ItemClick(msg.id))
    }
}
```

Reducer

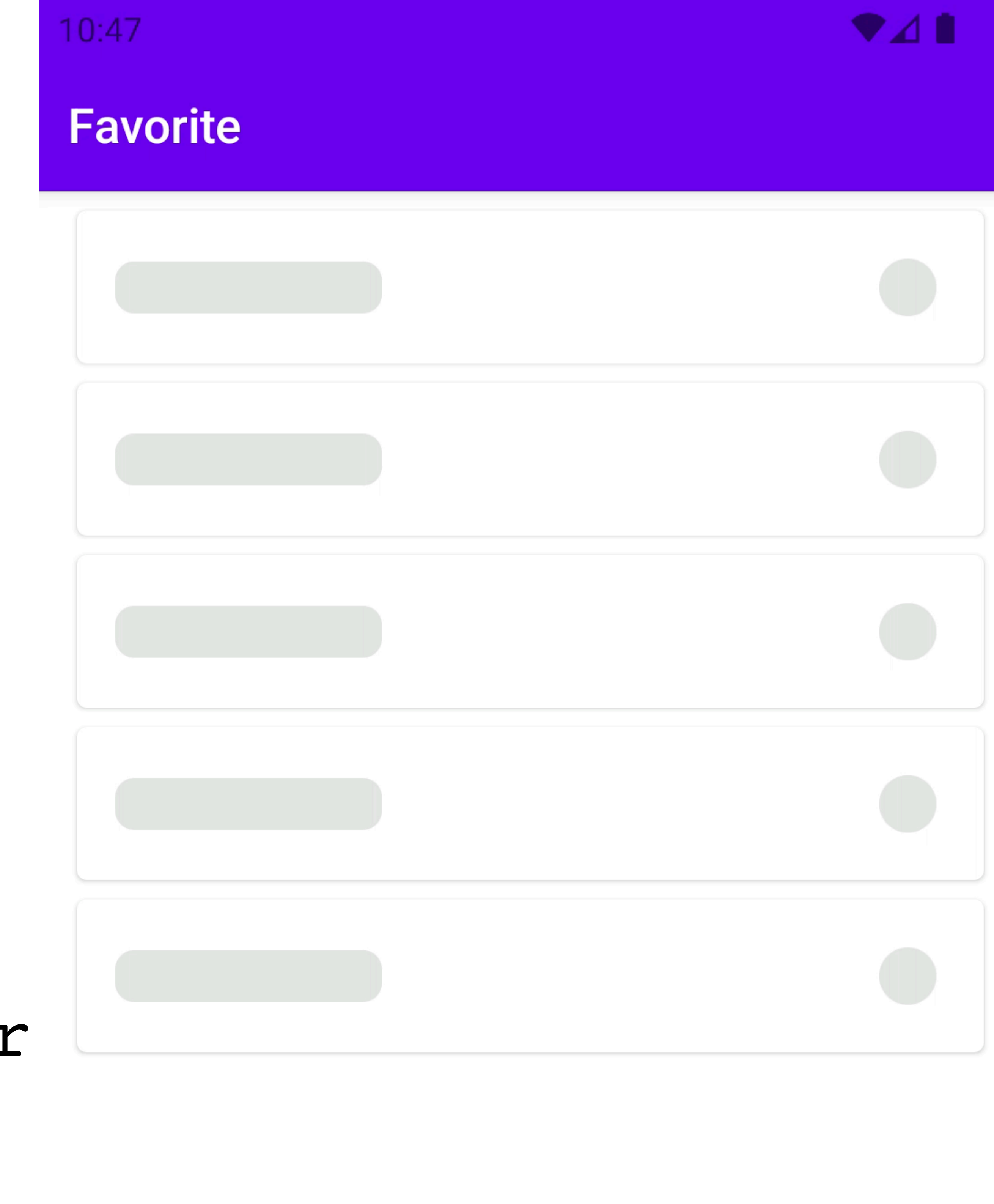
```
private fun ResultBuilder<State, Eff>.outerReducer(msg: Msg.Outer) {
    when (msg) {
        is Msg.Outer.RemoveFavorite -> ...
        is Msg.Outer.RetryLoad -> {
            if (state.content is LCE.Error && !state.content.inProgress) {
                state { State(LCE.Loading()) }
                eff(Eff.Inner.LoadFav)
            }
        }
        is Msg.Outer.ItemClick -> eff(Eff.Outer.ItemClick(msg.id))
    }
}
```

Eff

```kotlin
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```

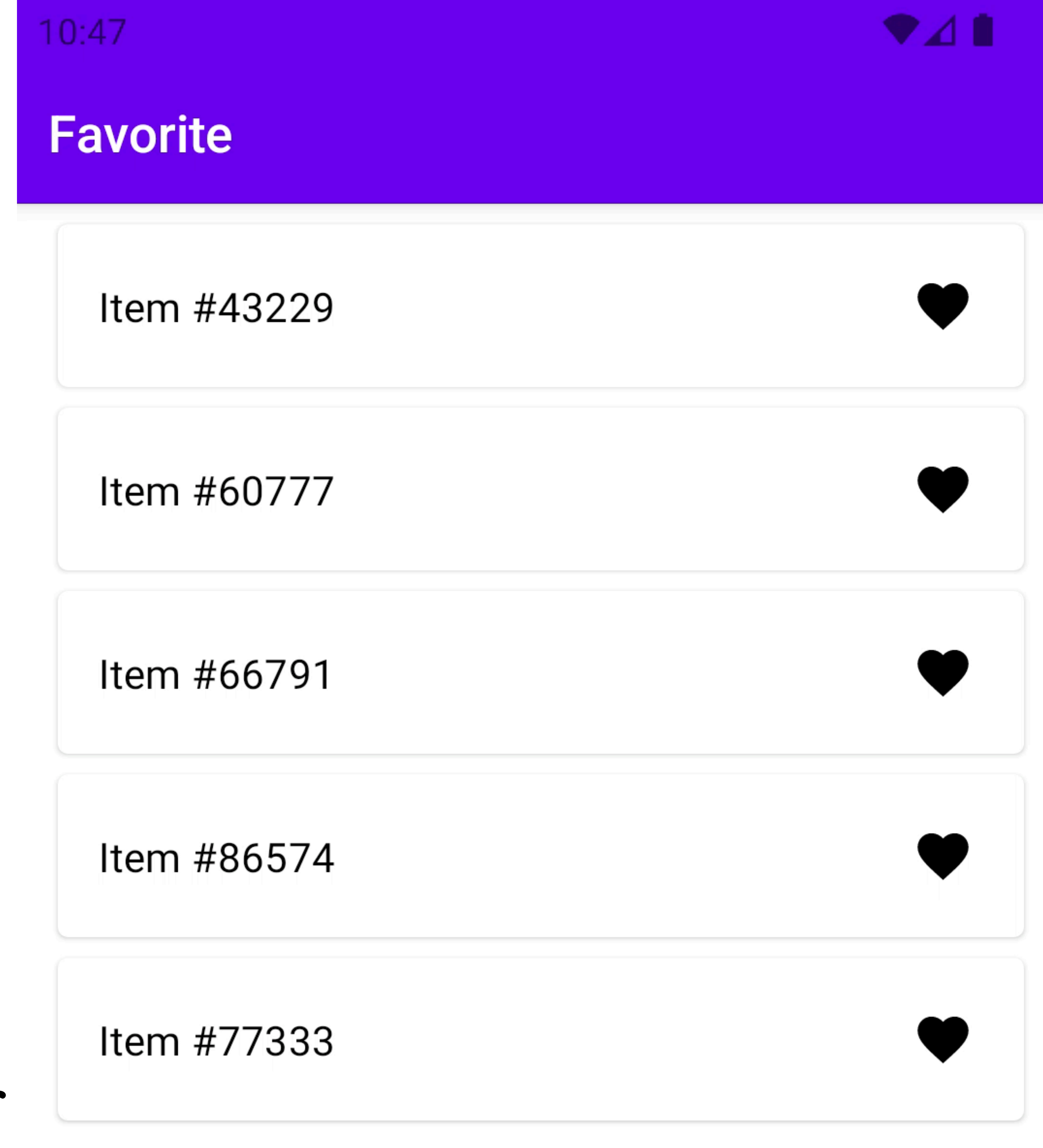


```kotlin
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```



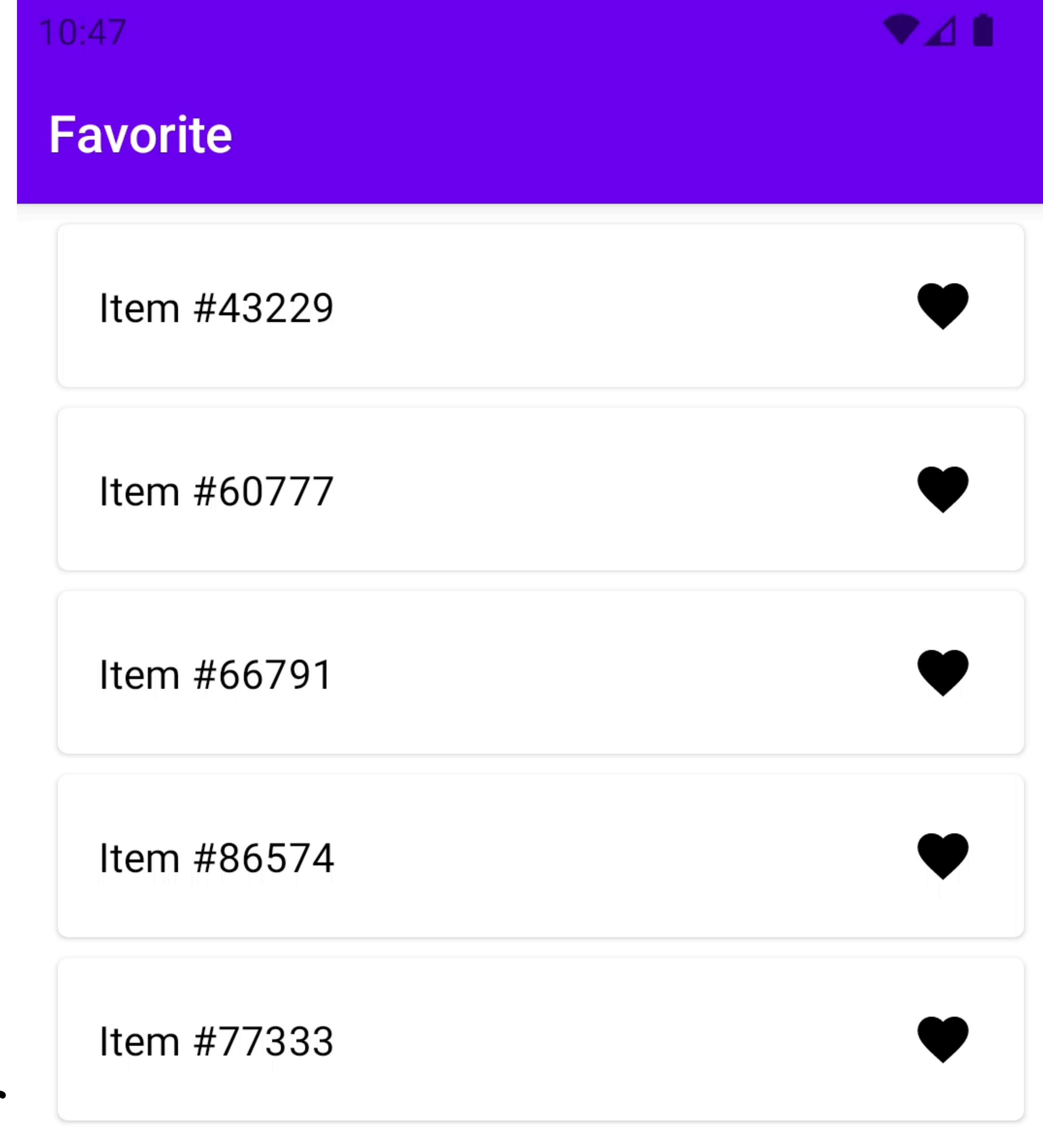
Eff

```kotlin
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```



Eff

```
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```



Eff

```
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```

10:47

Favorite

Item #43229

Item #60777

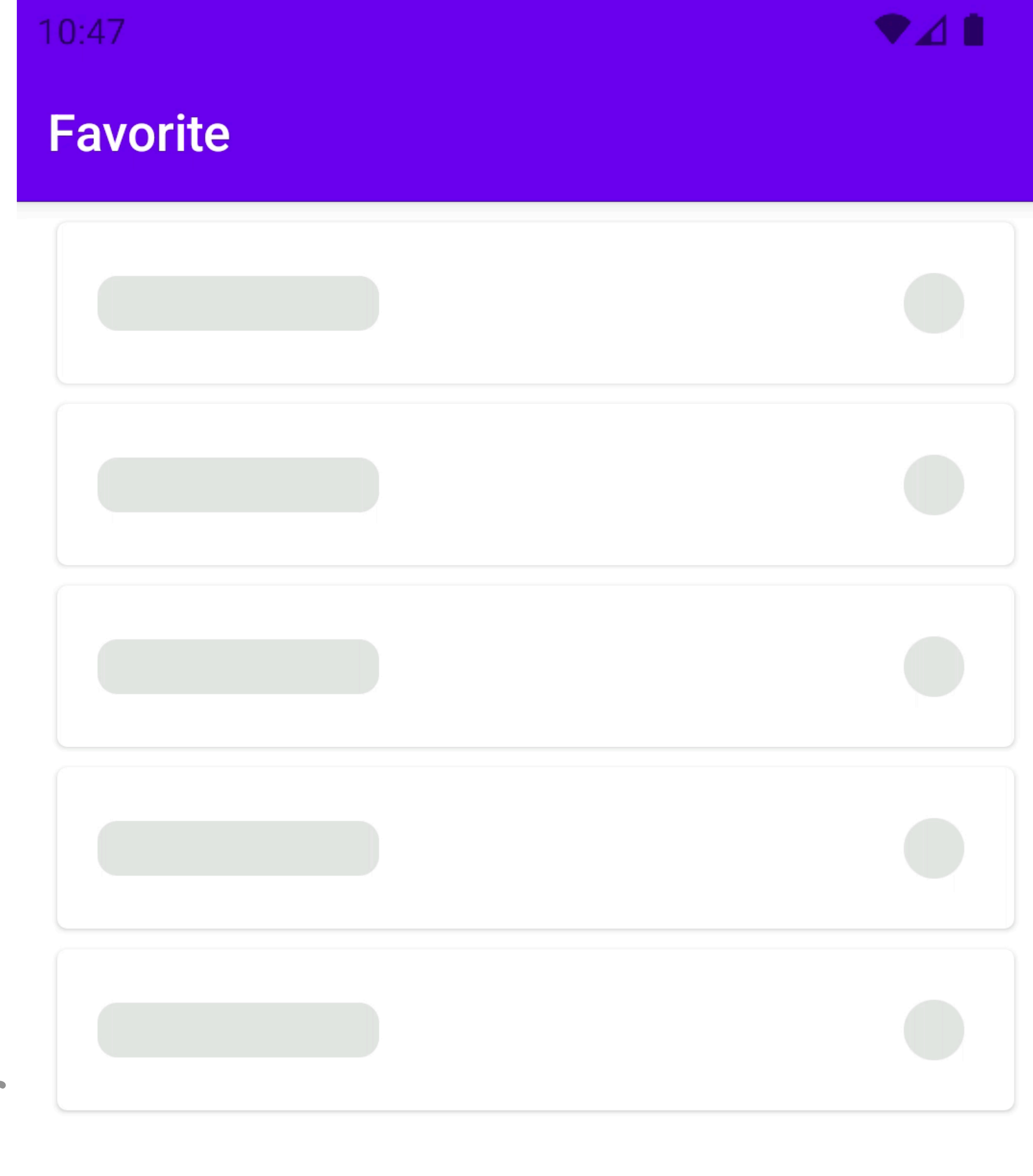
Item #66791

Item #86574

Item #77333

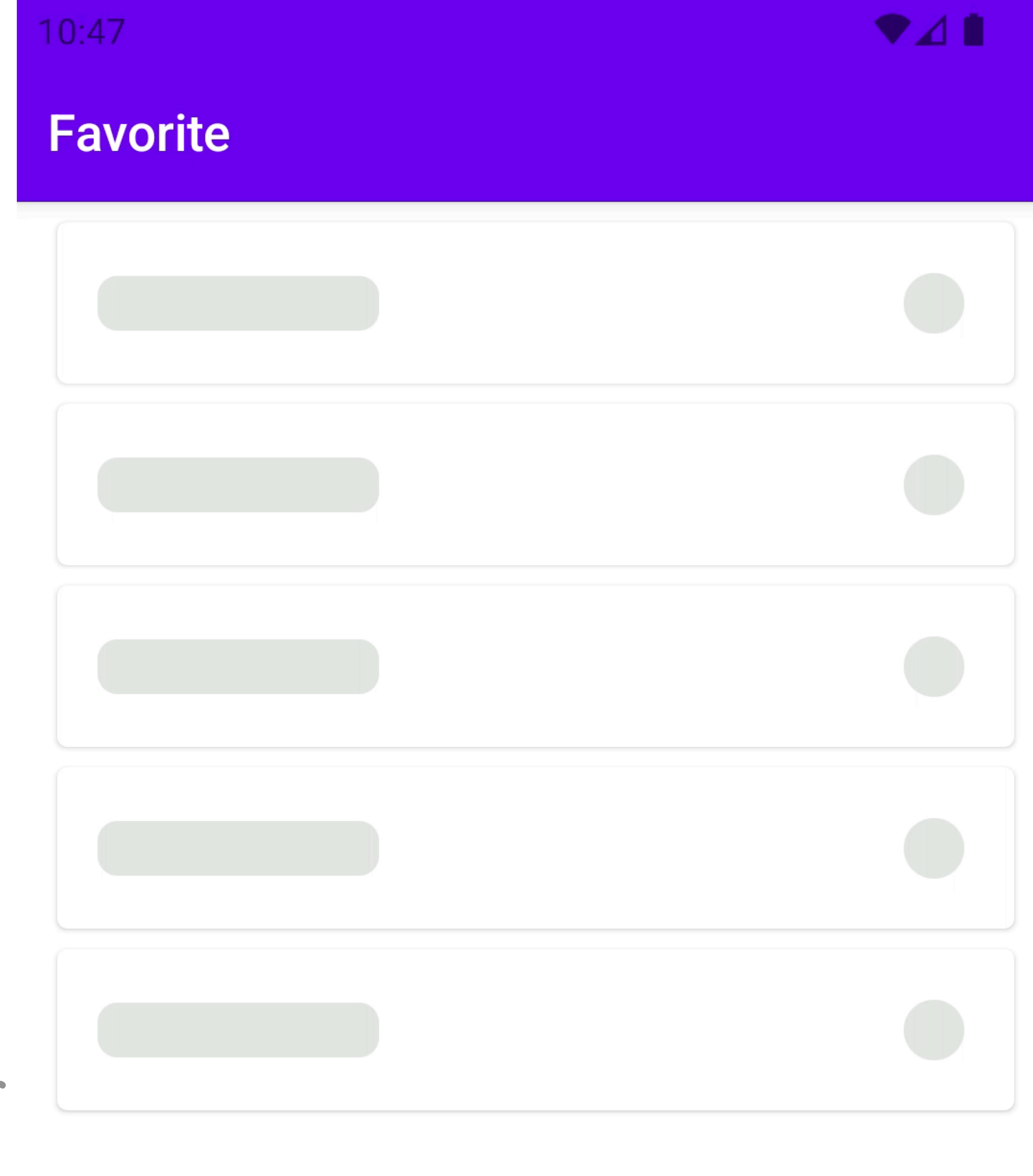
Show snackbar

```
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```



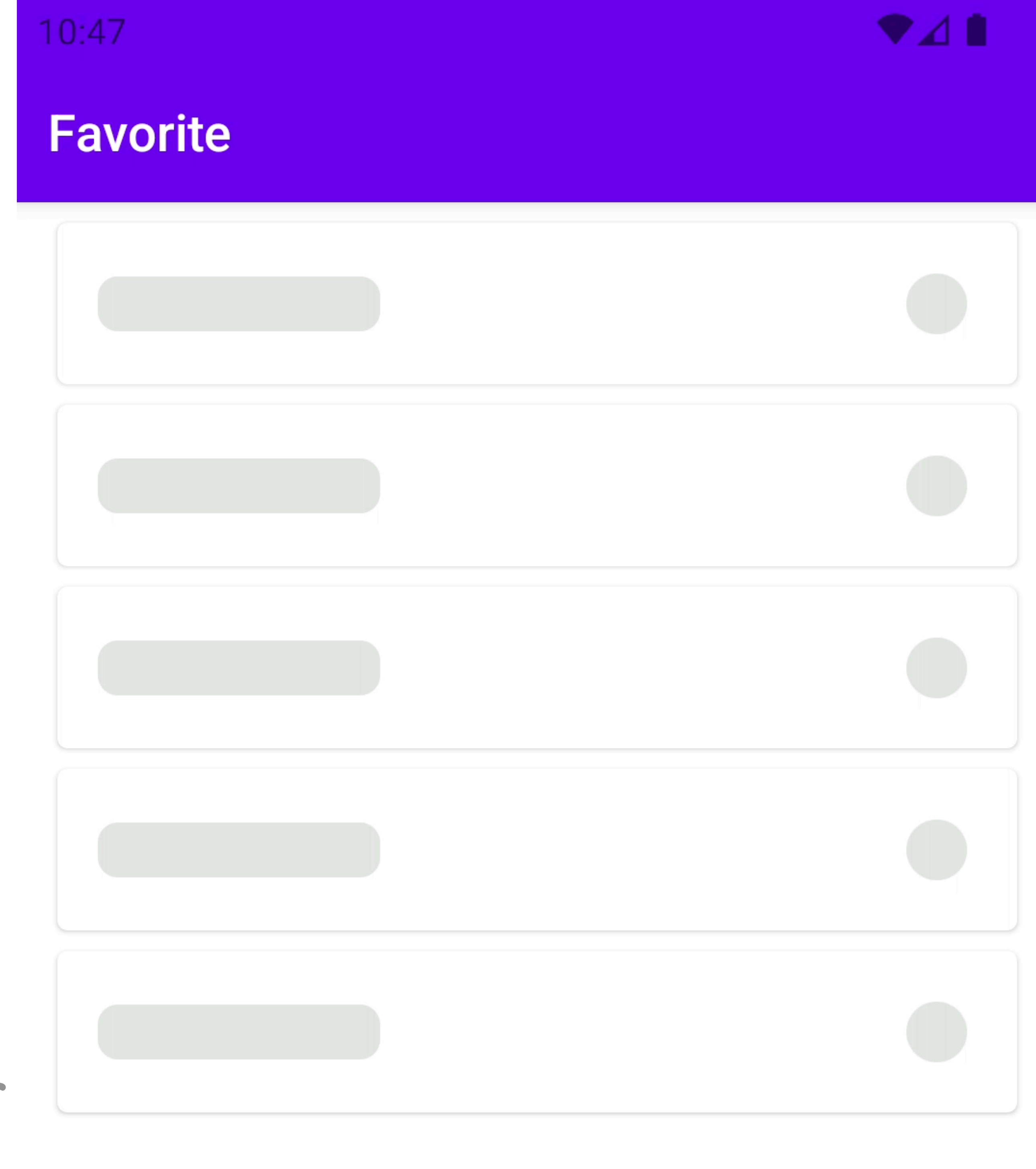
Eff

```kotlin
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```



Eff

```
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```



Internal interaction

EffectHandler

```kotlin
internal class FavoriteEffHandler(
    private val repository: FavoriteRepository,
) : EffectHandler<Eff.Inner, Msg.Inner> {

    override fun handleEff(eff: Eff.Inner): Flow<Msg.Inner> = when (eff) {
        is Eff.Inner.LoadFav -> flow {
            ...
        }
        is Eff.Inner.RemoveItem -> flow {
            ...
        }
        Eff.Inner.ObserveFavUpdates -> repository.newFavoriteSource().map {
            ...
        }
    }

}
```

EffectHandler

```
internal class FavoriteEffHandler(
    private val repository: FavoriteRepository,
) : EffectHandler<Eff.Inner, Msg.Inner> {

    override fun handleEff(eff: Eff.Inner): Flow<Msg.Inner> = when (eff) {
        is Eff.Inner.LoadFav -> flow {
            emit(
                Msg.Inner.ItemLoadingResult(repository.loadFavoriteItems())
            )
        }
        is Eff.Inner.RemoveItem -> flow {
            emit(
                repository.removeFavoriteItem(eff.id).fold(
                    onSuccess = { Msg.Inner.ItemRemoveResult.Done(eff.id) },
                    onFailure = { Msg.Inner.ItemRemoveResult.Error(eff.id, null) }
                )
            )
        }
        Eff.Inner.ObserveFavUpdates -> repository.newFavoriteSource().map {
            Msg.Inner.AddItem(it)
        }
```

EffectHandler

```
internal class FavoriteEffHandler(
    private val repository: FavoriteRepository,
) : EffectHandler<Eff.Inner, Msg.Inner> {

    override fun handleEff(eff: Eff.Inner): Flow<Msg.Inner> = when (eff) {
        is Eff.Inner.LoadFav -> flow {
            emit(
                Msg.Inner.ItemLoadingResult(repository.loadFavoriteItems())
            )
        }
        is Eff.Inner.RemoveItem -> flow {
            emit(
                repository.removeFavoriteItem(eff.id).fold(
                    onSuccess = { Msg.Inner.ItemRemoveResult.Done(eff.id) },
                    onFailure = { Msg.Inner.ItemRemoveResult.Error(eff.id, null) }
                )
            )
        }
        Eff.Inner.ObserveFavUpdates -> repository.newFavoriteSource().map {
            Msg.Inner.AddItem(it)
        }
```

EffectHandler

```
internal class FavoriteEffHandler(
    private val repository: FavoriteRepository,
) : EffectHandler<Eff.Inner, Msg.Inner> {

    override fun handleEff(eff: Eff.Inner): Flow<Msg.Inner> = when (eff) {
        is Eff.Inner.LoadFav -> flow {
            emit(
                Msg.Inner.ItemLoadingResult(repository.loadFavoriteItems())
            )
        }
        is Eff.Inner.RemoveItem -> flow {
            emit(
                repository.removeFavoriteItem(eff.id).fold(
                    onSuccess = { Msg.Inner.ItemRemoveResult.Done(eff.id) },
                    onFailure = { Msg.Inner.ItemRemoveResult.Error(eff.id, null) }
                )
            )
        }
        Eff.Inner.ObserveFavUpdates -> repository.newFavoriteSource().map {
            Msg.Inner.AddItem(it)
        }
```

EffectHandler

```
internal class FavoriteEffHandler(
    private val repository: FavoriteRepository,
) : EffectHandler<Eff.Inner, Msg.Inner> {

    override fun handleEff(eff: Eff.Inner): Flow<Msg.Inner> = when (eff) {
        is Eff.Inner.LoadFav -> flow {
            emit(
                Msg.Inner.ItemLoadingResult(repository.loadFavoriteItems())
            )
        }
        is Eff.Inner.RemoveItem -> flow {
            emit(
                repository.removeFavoriteItem(eff.id).fold(
                    onSuccess = { Msg.Inner.ItemRemoveResult.Done(eff.id) },
                    onFailure = { Msg.Inner.ItemRemoveResult.Error(eff.id, null) }
                )
            )
        }
        Eff.Inner.ObserveFavUpdates -> repository.newFavoriteSource().map {
            Msg.Inner.AddItem(it)
        }
```

External subscription

In EffectHandler

FavoriteListStore

```kotlin
internal class FavoriteListStore(
    favoriteEffectHandler: FavoriteEffectHandler,
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteListStore",
    reducer = FavoriteFeature.reducer,
    initialState = State(LCE.Loading()),
    initialEffects = setOf(Eff.Inner.LoadFav, Eff.Inner.ObserveFavUpdates),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

impl/FavoriteListFeature.kt

FavoriteListStore

```
internal class FavoriteListStore(
    favoriteEffectHandler: FavoriteEffectHandler,
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteListStore",
    reducer = FavoriteFeature.reducer,
    initialState = State(LCE.Loading()),
    initialEffects = setOf(Eff.Inner.LoadFav, Eff.Inner.ObserveFavUpdates),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

FavoriteListStore

```
internal class FavoriteListStore(
    favoriteEffectHandler: FavoriteEffectHandler,
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteListStore",
    reducer = FavoriteFeature.reducer,
    initialState = State(LCE.Loading()),
    initialEffects = setOf(Eff.Inner.LoadFav, Eff.Inner.ObserveFavUpdates),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

FavoriteListStore

```
internal class FavoriteListStore(
    favoriteEffectHandler: FavoriteEffectHandler,
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteListStore",
    reducer = FavoriteFeature.reducer,
    initialState = State(LCE.Loading()),
    initialEffects = setOf(Eff.Inner.LoadFav, Eff.Inner.ObserveFavUpdates),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

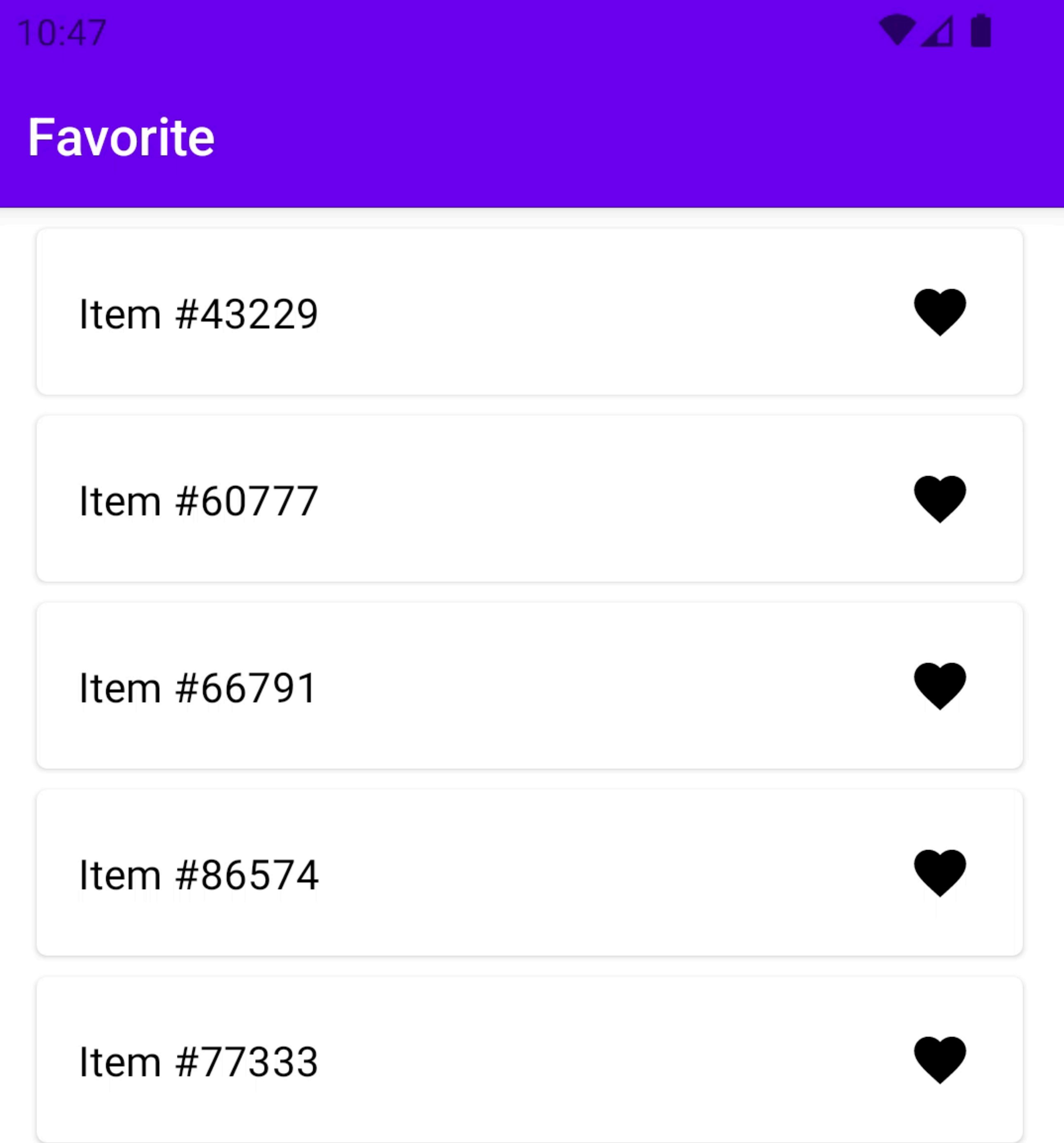
FavoriteListStore

```
internal class FavoriteListStore(
    favoriteEffectHandler: FavoriteEffectHandler,
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteListStore",
    reducer = FavoriteFeature.reducer,
    initialState = State(LCE.Loading()),
    initialEffects = setOf(Eff.Inner.LoadFav, Eff.Inner.ObserveFavUpdates),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

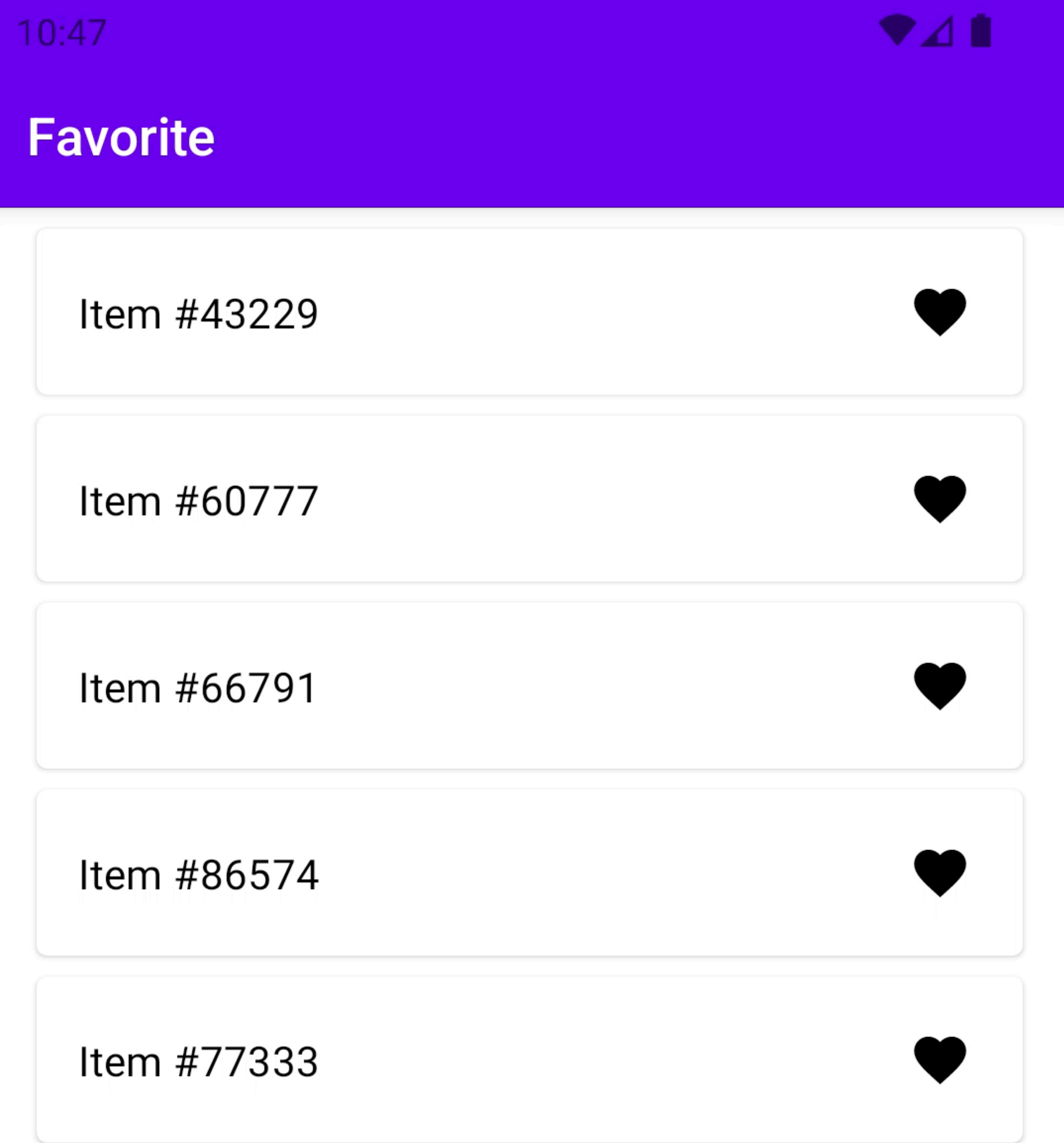
FavoriteListStore

```
internal class FavoriteListStore(
    favoriteEffectHandler: FavoriteEffectHandler,
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteListStore",
    reducer = FavoriteListFeature.reducer,
    initialState = State(LCE.Loading()),
    initialEffects = setOf(Eff.Inner.LoadFav, Eff.Inner.ObserveFavUpdates),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

Run it!



Run it!



We’ve build up

Simple favorite list

Let’s improve it with
Pagination!

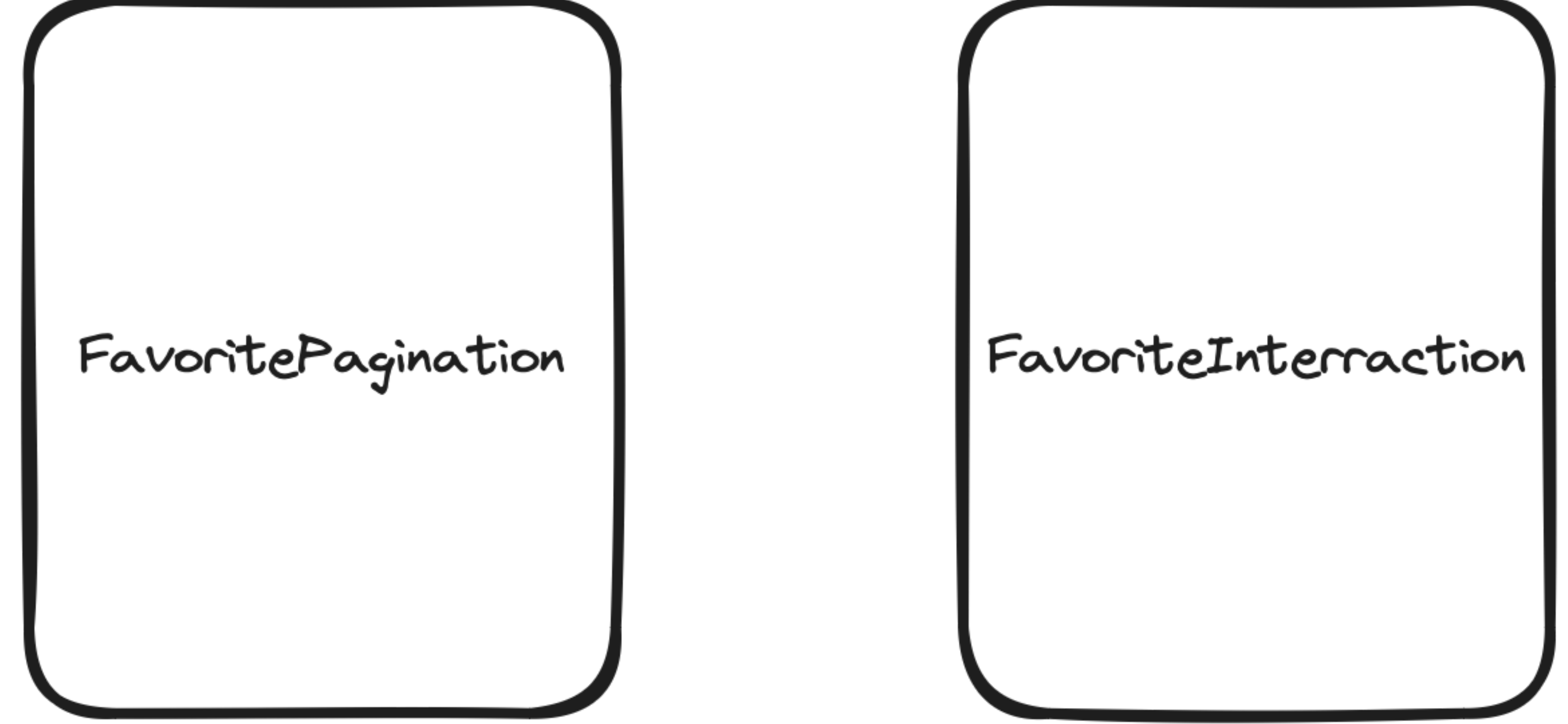
IDK what to do

Implement pagination in Favorite feature

Build up reusable pagination
Implement pagination in Favorite feature

How to build reusable component?

FavoriteStore

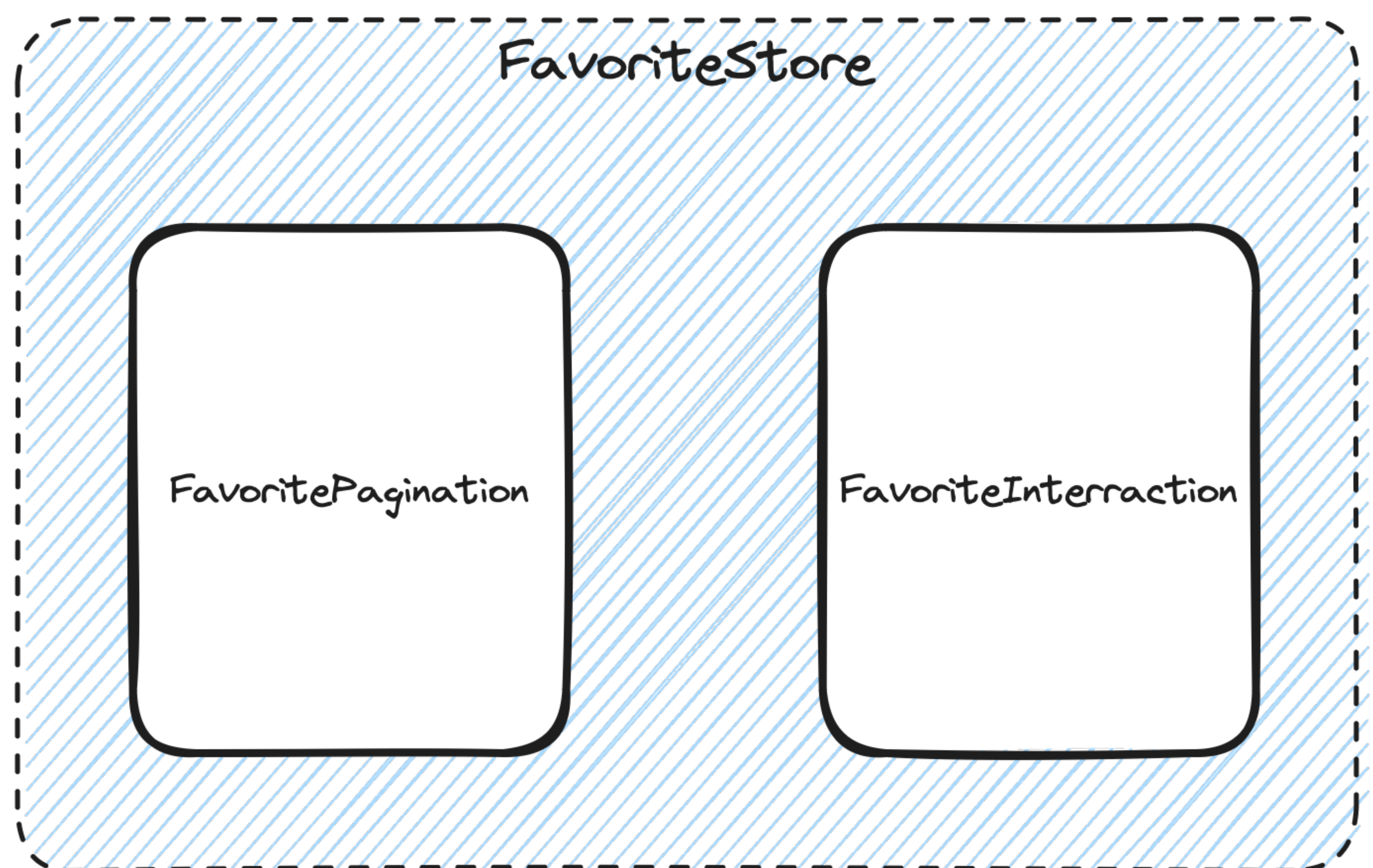

FavoritePagination
FavoriteInteraction

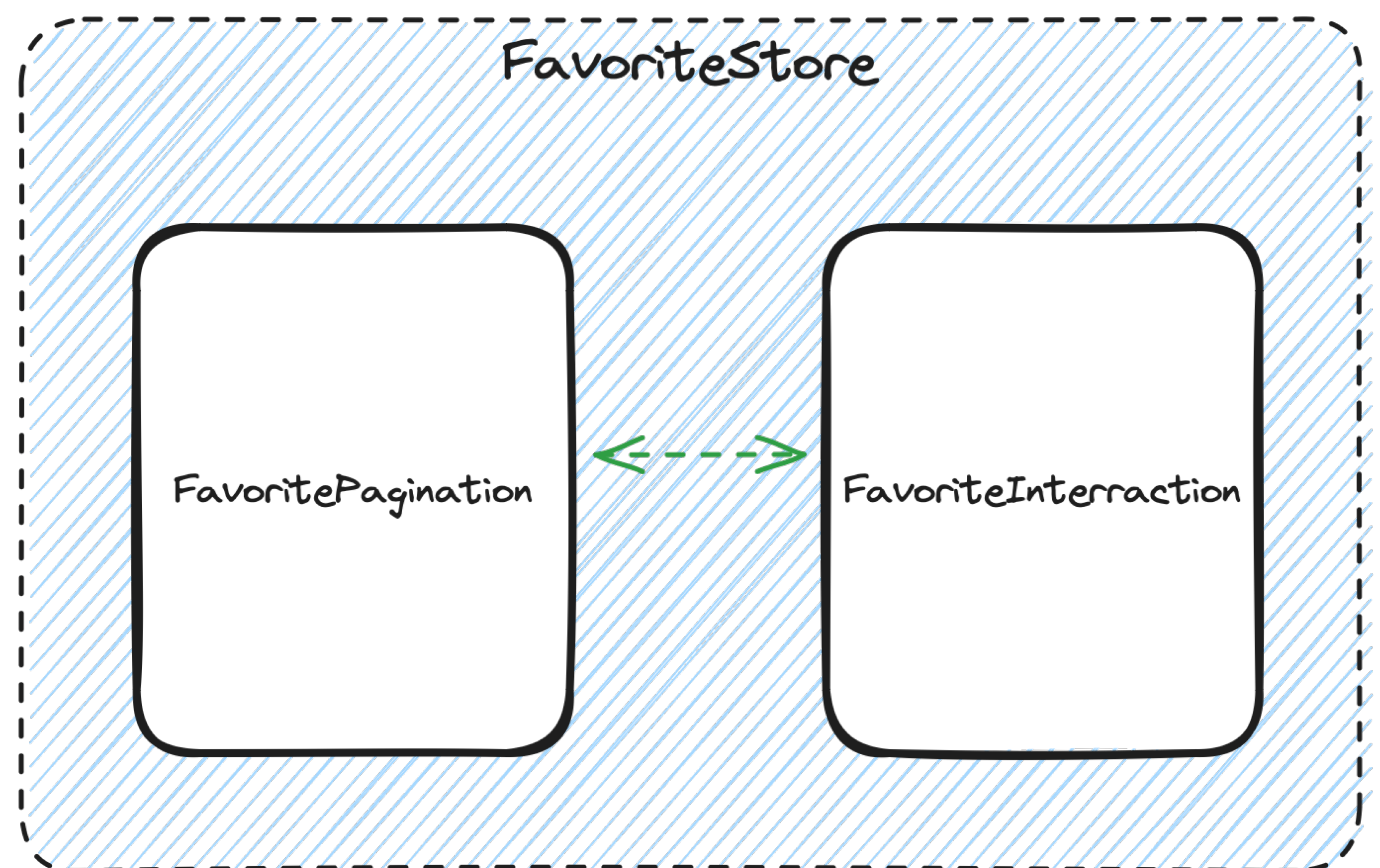
FavoritePagination

FavoriteInterraction

favorite list + pagination logic

Update favorite status logic


FavoriteStore
FavoritePagination
FavoriteInteraction


FavoriteStore
FavoritePagination
FavoriteInteraction

https://badoo.github.io/MVICore/bestpractices/f2f/

https://badoo.github.io/MVICore/bestpractices/f2f/

Code it!

Aggregate favorite

```
internal class FavoriteAggregatorStore(
    private val paginationStore: FavoritePaginationStore,
    private val interactionStore: FavoriteInteractionStore,
) : AggregatorStore<Msg, State, Eff>(
    name = "FavoriteAggregatorStore"
)
```

impl/aggregated/FavoriteAggregatorStore.kt

FavoriteInteractionStore

```kotlin
internal class FavoriteInteractionStore(
    favoriteEffectHandler: FavoriteEffectHandler,
    favoriteAnalytics: FavoriteAnalytics
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteInteraction",
    initialState = State(emptyMap()),
    reducer = reducer,
    initialEffects = setOf(Eff.Inner.ObserveFavUpdates),
    effectHandlers = listOf(favoriteEffectHandler.adaptCast())
)
```

impl/aggregated/FavoriteInteractionStore.kt

FavoriteInteractionStore: Msg

```
sealed interface Msg {

    sealed interface Outer : Msg {
        data class ItemClick(val id: String) : Outer
        data class UpdateFavorite(val id: String, val isFavorite: Boolean) : Outer
    }

    sealed interface Inner : Msg {

        data class AddItem(val item: FavoriteItem) : Inner
        sealed interface ItemRemoveResult : Inner {
            ...
        }
    }
}
```

FavoriteInteractionStore: State

```
data class State(
    /**
     * Contains currently updating items with desired favorite value.
     */
    val updatingItems: Map<String, FavoriteUpdate>
)
```

FavoriteInteractionStore: Eff

```
sealed interface Eff {

    sealed interface Outer : Eff {
        sealed interface ItemUpdate : Outer {
            data class Started(override val item: FavoriteUpdate) : ItemUpdate
            data class Finished(override val item: FavoriteUpdate) : ItemUpdate
            data class Error(override val item: FavoriteUpdate, val throwable: Throwable?)
                                                                : ItemUpdate
        }
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```

FavoritePaginationStore

```
internal class FavoritePaginationStore(
    reducerStoreFactory: ReducerStoreFactory,
    dataFetcher: DataFetcher,
) : PaginationStore<FavoriteItem>(
    name = "FavoritePagination",
    reducerStoreFactory = reducerStoreFactory,
    dataFetcher = dataFetcher

)
```

impl/aggregated/FavoritePaginationStore.kt

PaginationStore

```
open class PaginationStore<Item>(
    name: String,
    reducerStoreFactory: ReducerStoreFactory,
    dataFetcher: PaginationDataFetcher<Item>,
    pageSize: Int = PaginationFeature.DEFAULT_PAGE_SIZE,
) : Store<PaginationMsg<Item>, PaginationState<Item>, PaginationEff> by
reducerStoreFactory.create(
    name = name,
    initialState = PaginationState.Initial(pageSize),
    reducer = PaginationFeature.reducer(),
    initialEffects = PaginationEff.Initial(pageSize),
    PaginationEffectHandler(dataFetcher).adaptCast()
)
```

sample/core/pagination/src/commonMain/kotlin/io/github/ikarenkov/sample/core/pagination/PaginationFeature.kt

Aggregate favorite

```
internal object FavoriteAggregatedFeature {
    data class State(
        val pagination: PaginationState<FavoriteItem>,
        val favoriteInteraction: FavoriteInteractionFeature.State
    )

    sealed interface Msg {
        data class Pagination(val msg: PaginationMsg.Outer<FavoriteItem>) : Msg
        data class FavoriteInteraction(val msg: FavoriteInteractionFeature.Msg.Outer) : Msg
    }

    sealed interface Eff {
        data class FavoriteInteraction(val eff: FavoriteInteractionFeature.Eff.Outer) : Eff
    }
}
```

Aggregate favorite

```
internal object FavoriteAggregatedFeature {
    data class State(
        val pagination: PaginationState<FavoriteItem>,
        val favoriteInteraction: FavoriteInteractionFeature.State
    )

    sealed interface Msg {
        data class Pagination(val msg: PaginationMsg.Outer<FavoriteItem>) : Msg
        data class FavoriteInteraction(val msg: FavoriteInteractionFeature.Msg.Outer) : Msg
    }

    sealed interface Eff {
        data class FavoriteInteraction(val eff: FavoriteInteractionFeature.Eff.Outer) : Eff
    }
}
```

Aggregate favorite

```
internal object FavoriteAggregatedFeature {
    data class State(
        val pagination: PaginationState<FavoriteItem>,
        val favoriteInteraction: FavoriteInteractionFeature.State
    )

    sealed interface Msg {
        data class Pagination(val msg: PaginationMsg.Outer<FavoriteItem>) : Msg
        data class FavoriteInteraction(val msg: FavoriteInteractionFeature.Msg.Outer) : Msg
    }

    sealed interface Eff {
        data class FavoriteInteraction(val eff: FavoriteInteractionFeature.Eff.Outer) : Eff
    }
}
```

Aggregate favorite

```
internal object FavoriteAggregatedFeature {
    data class State(
        val pagination: PaginationState<FavoriteItem>,
        val favoriteInteraction: FavoriteInteractionFeature.State
    )

    sealed interface Msg {
        data class Pagination(val msg: PaginationMsg.Outer<FavoriteItem>) : Msg
        data class FavoriteInteraction(val msg: FavoriteInteractionFeature.Msg.Outer) : Msg
    }

    sealed interface Eff {
        data class FavoriteInteraction(val eff: FavoriteInteractionFeature.Eff.Outer) : Eff
    }
}
```

Aggregate favorite

```
override val state: StateFlow<State> =
    combine(paginationStore.state, interactionStore.state, ::State)
        .stateIn(
            scope = coroutineScope,
            started = SharingStarted.Lazily,
            initialValue = State(paginationStore.state.value, interactionStore.state.value)
        )

override val effects: Flow<Eff> = interactionStore.effects
    .filterIsInstance<FavoriteInteractionFeature.Eff.Outer>()
    .map { Eff.FavoriteInteraction(it) }

override fun accept(msg: Msg) {
    when (msg) {
        is Msg.FavoriteInteraction -> interactionStore.accept(msg.msg)
        is Msg.Pagination -> paginationStore.accept(msg.msg)
    }
}
```

Aggregate favorite

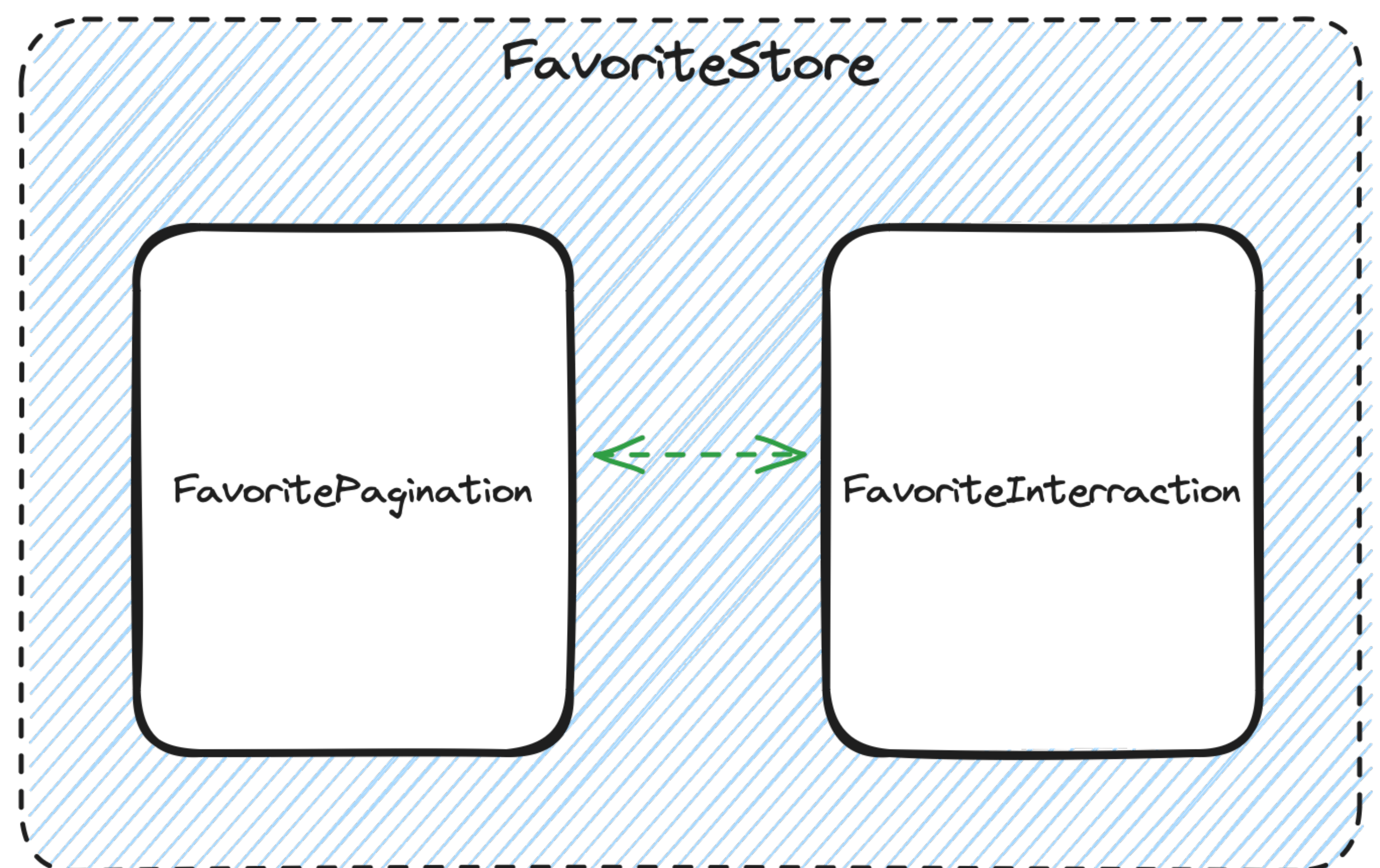
```
override val state: StateFlow<State> =
    combine(paginationStore.state, interactionStore.state, ::State)
        .stateIn(
            scope = coroutineScope,
            started = SharingStarted.Lazily,
            initialValue = State(paginationStore.state.value, interactionStore.state.value)
        )

override val effects: Flow<Eff> = interactionStore.effects
    .filterIsInstance<FavoriteInteractionFeature.Eff.Outer>()
    .map { Eff.FavoriteInteraction(it) }

override fun accept(msg: Msg) {
    when (msg) {
        is Msg.FavoriteInteraction -> interactionStore.accept(msg.msg)
        is Msg.Pagination -> paginationStore.accept(msg.msg)
    }
}
```

Aggregate favorite

```
override val state: StateFlow<State> =
    combine(paginationStore.state, interactionStore.state, ::State)
        .stateIn(
            scope = coroutineScope,
            started = SharingStarted.Lazily,
            initialValue = State(paginationStore.state.value, interactionStore.state.value)
        )

override val effects: Flow<Eff> = interactionStore.effects
    .filterIsInstance<FavoriteInteractionFeature.Eff.Outer>()
    .map { Eff.FavoriteInteraction(it) }

override fun accept(msg: Msg) {
    when (msg) {
        is Msg.FavoriteInteraction -> interactionStore.accept(msg.msg)
        is Msg.Pagination -> paginationStore.accept(msg.msg)
    }
}
```

Aggregate favorite

```kotlin
override val state: StateFlow<State> =
    combine(paginationStore.state, interactionStore.state, ::State)
        .stateIn(
            scope = coroutineScope,
            started = SharingStarted.Lazily,
            initialValue = State(paginationStore.state.value, interactionStore.state.value)
        )

override val effects: Flow<Eff> = interactionStore.effects
    .filterIsInstance<FavoriteInteractionFeature.Eff.Outer>()
    .map { Eff.FavoriteInteraction(it) }

override fun accept(msg: Msg) {
    when (msg) {
        is Msg.FavoriteInteraction -> interactionStore.accept(msg.msg)
        is Msg.Pagination -> paginationStore.accept(msg.msg)
    }
}
```


FavoriteStore
FavoritePagination
FavoriteInteraction

Store to store binding

```
init {
    bindEffToMsg(
        coroutineScope,
        interactionStore,
        paginationStore
    ) { eff ->
        when (eff) {
            is FavoriteInteractionFeature.Eff.Outer.ItemAdded ->
                PaginationMsg.Outer.AddItem(eff.item)
            is FavoriteInteractionFeature.Eff.Inner.RemoveItem -> ...
            is FavoriteInteractionFeature.Eff.Outer.ItemRemoveError -> ...
            else -> null
        }
    }
}
```

Store to store binding

```
init {
    bindEffToMsg(
        coroutineScope,
        interactionStore,
        paginationStore
    ) { eff ->
        when (eff) {
            is FavoriteInteractionFeature.Eff.Outer.ItemAdded ->
                PaginationMsg.Outer.AddItem(eff.item)
            is FavoriteInteractionFeature.Eff.Inner.RemoveItem -> ...
            is FavoriteInteractionFeature.Eff.Outer.ItemRemoveError -> ...
            else -> null
        }
    }
}
```

Store to store binding

```
init {
    bindEffToMsg(
        coroutineScope,
        interactionStore,
        paginationStore
    ) { eff ->
        when (eff) {
            is FavoriteInteractionFeature.Eff.Outer.ItemAdded ->
                PaginationMsg.Outer.AddItem(eff.item)
            is FavoriteInteractionFeature.Eff.Inner.RemoveItem -> ...
            is FavoriteInteractionFeature.Eff.Outer.ItemRemoveError -> ...
            else -> null
        }
    }
}
```

Store to store binding

```
init {
    bindEffToMsg(
        coroutineScope,
        interactionStore,
        paginationStore
    ) { eff ->
        when (eff) {
            is FavoriteInteractionFeature.Eff.Outer.ItemAdded ->
                PaginationMsg.Outer.AddItem(eff.item)
            is FavoriteInteractionFeature.Eff.Inner.RemoveItem -> ...
            is FavoriteInteractionFeature.Eff.Outer.ItemRemoveError -> ...
            else -> null
        }
    }
}
```

Store to store binding

```
init {
    bindEffToMsg(
        coroutineScope,
        interactionStore,
        paginationStore
    ) { eff ->
        when (eff) {
            is FavoriteInteractionFeature.Eff.Outer.ItemAdded ->
                PaginationMsg.Outer.AddItem(eff.item)
            is FavoriteInteractionFeature.Eff.Inner.RemoveItem -> ...
            is FavoriteInteractionFeature.Eff.Outer.ItemRemoveError -> ...
            else -> null
        }
    }
}
```

What kind of communication?

1. Remove from favorite → Pagination.updateItem
2. Add to favorite → Pagination.insertItem
3. Error removing from favorite → Pagination.update item
4. Success removal → Pagination.deleteItem

What kind of communication?

1. Remove from favorite → Pagination.updateItem
2. Add to favorite → Pagination.insertItem
3. Error removing from favorite → Pagination.update item
4. Success removal → Pagination.deleteItem

What kind of communication?

1. Remove from favorite → Pagination.updateItem
2. Add to favorite → Pagination.insertItem
3. Error removing from favorite → Pagination.update item
4. Success removal → Pagination.deleteItem

What kind of communication?

1. Remove from favorite → Pagination.updateItem
2. Add to favorite → Pagination.insertItem
3. Error removing from favorite → Pagination.update item
4. Success removal → Pagination.deleteItem

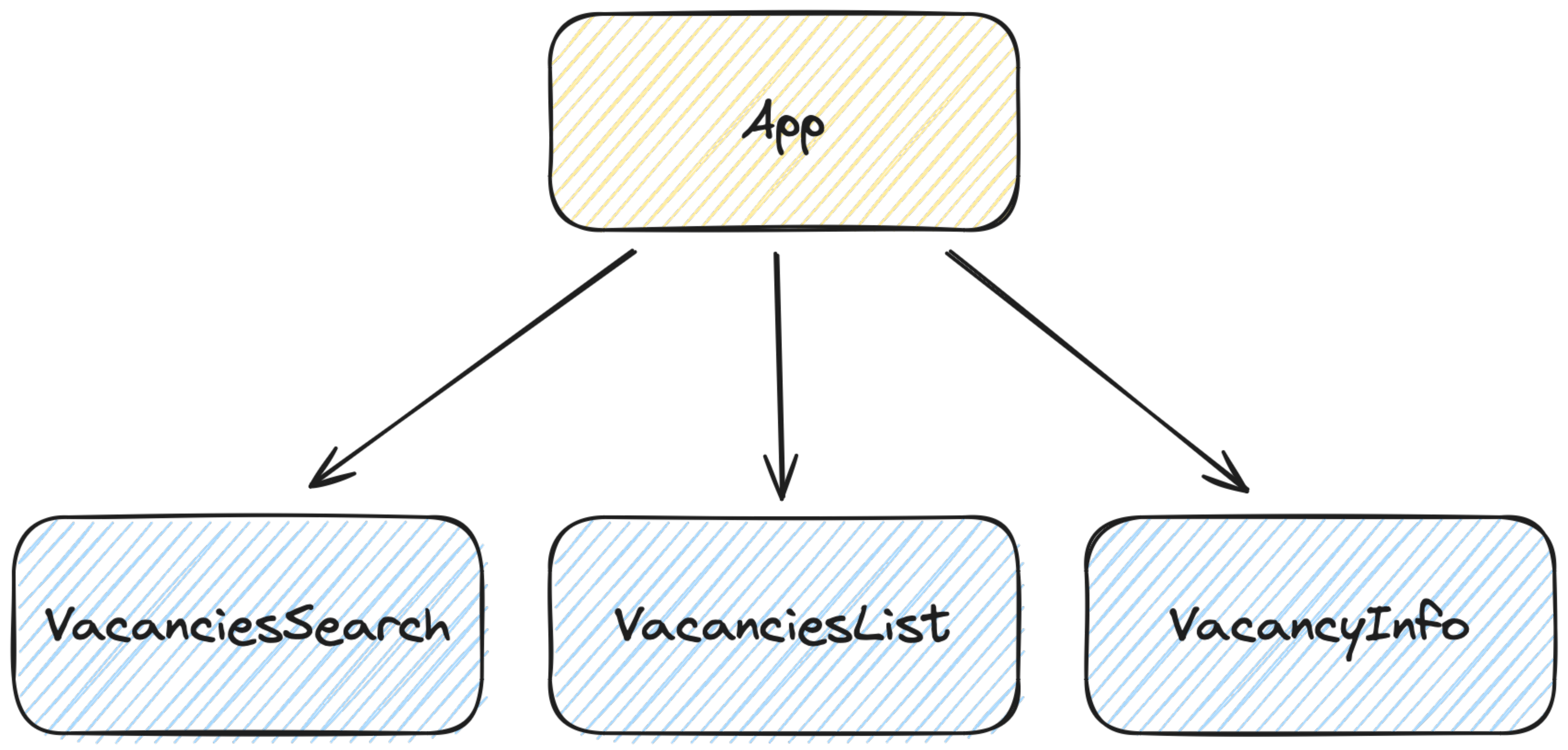
What kind of communication?

1. Remove from favorite → Pagination.updateItem
2. Add to favorite → Pagination.insertItem
3. Error removing from favorite → Pagination.update item
4. Success removal → Pagination.deleteItem

What we have learned

1. Build stores
2. Reuse features logic
3. Aggregate stores
4. Merge State, Msg, Eff

How to integrate
Favorite feature?

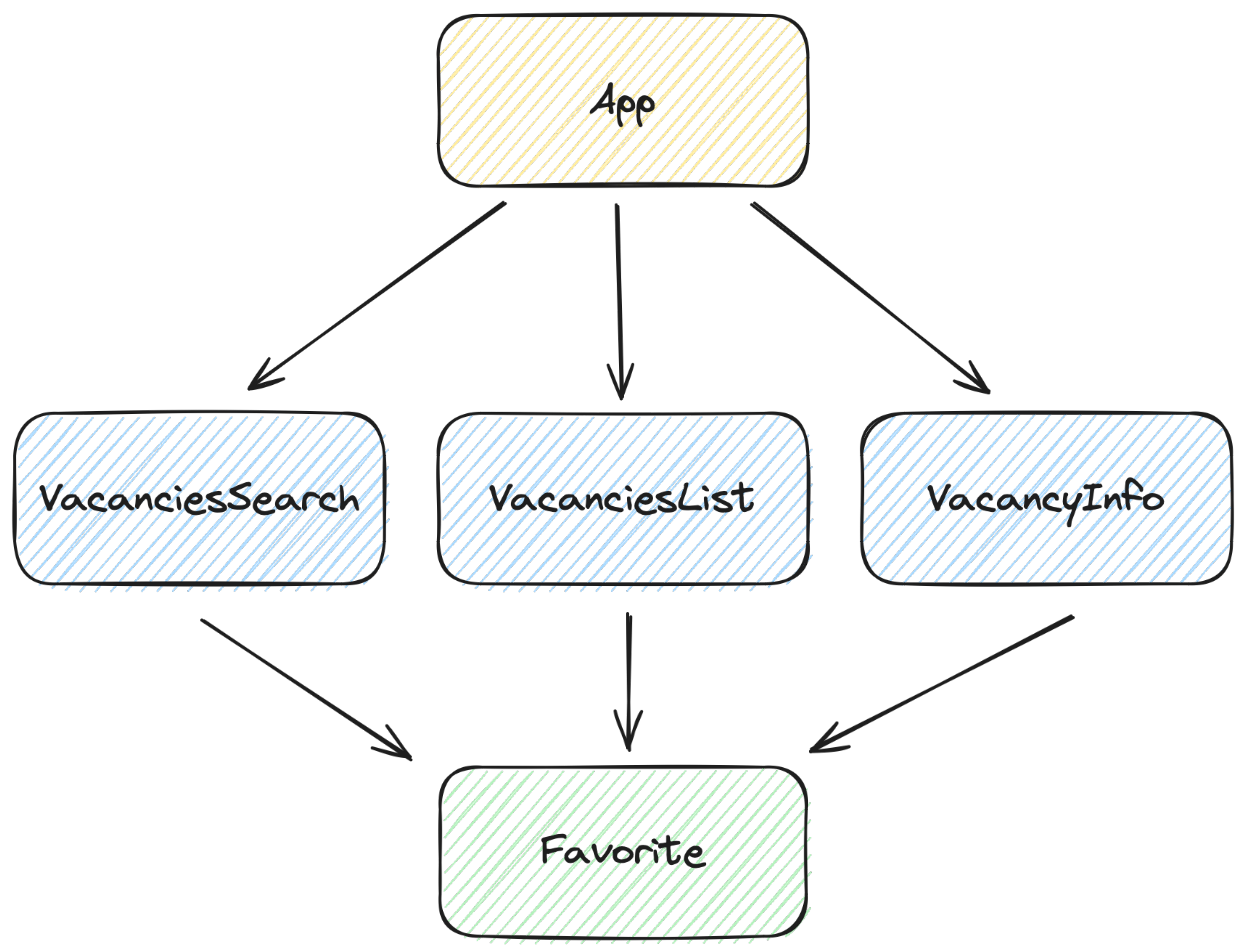

App
VacanciesSearch
VacanciesList
VacancyInfo

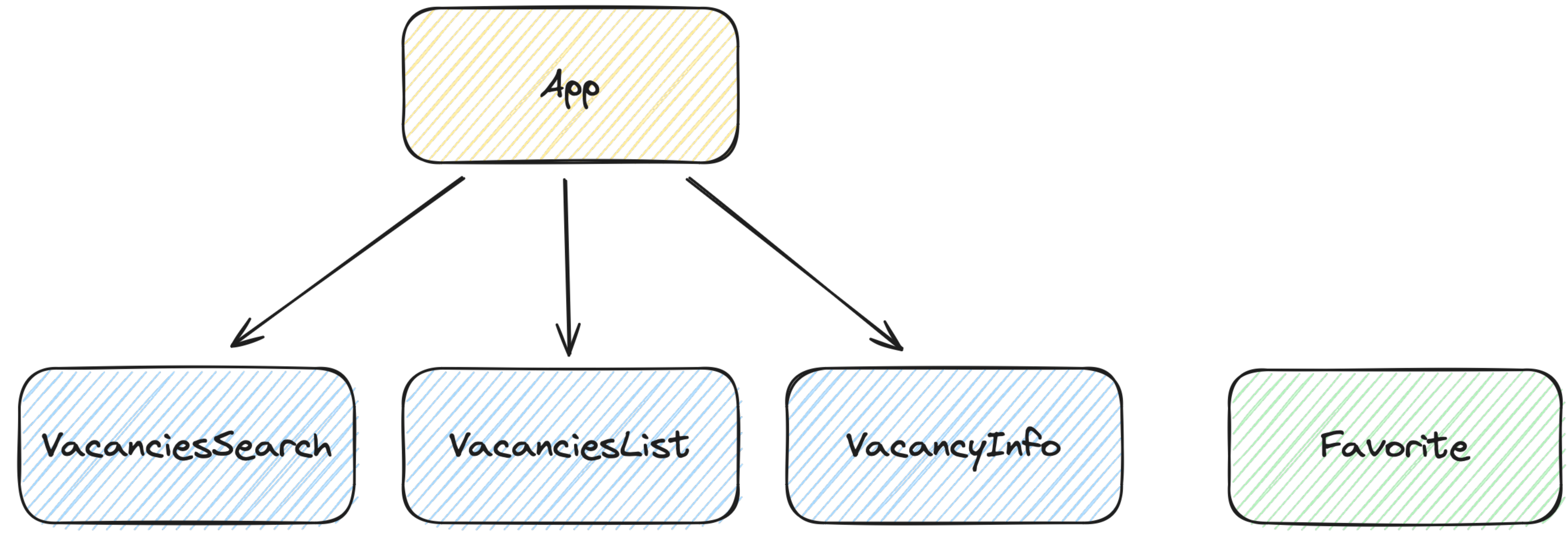
App

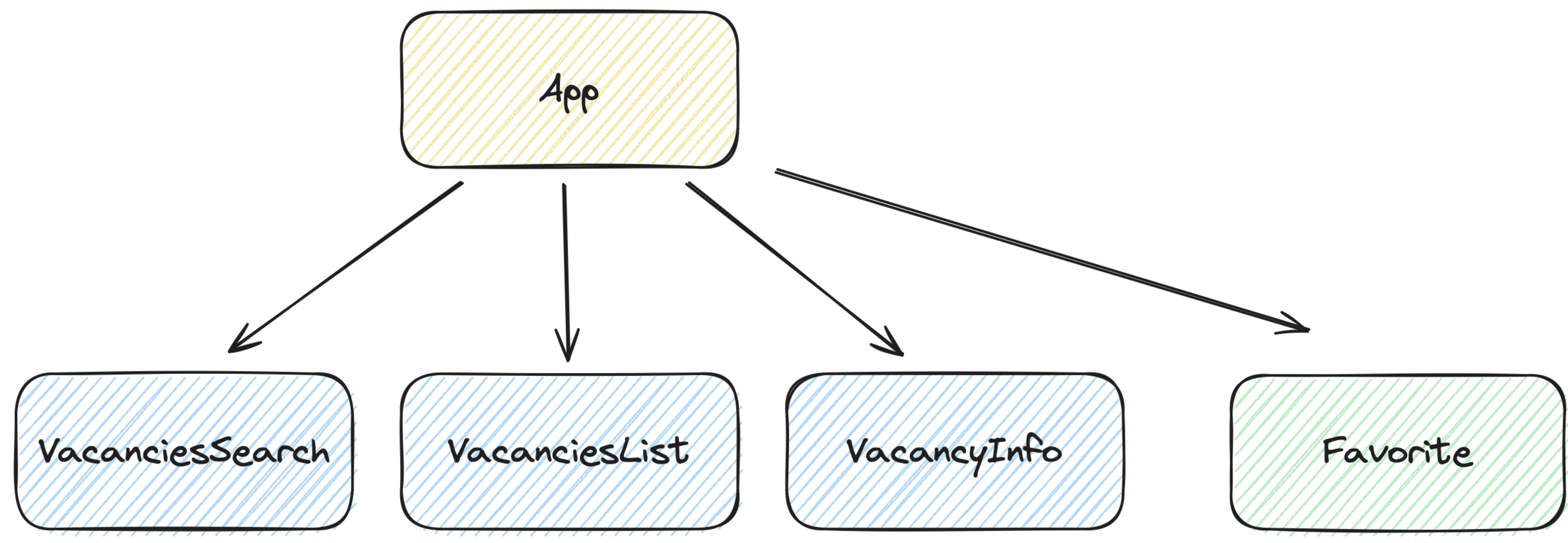
VacanciesSearch

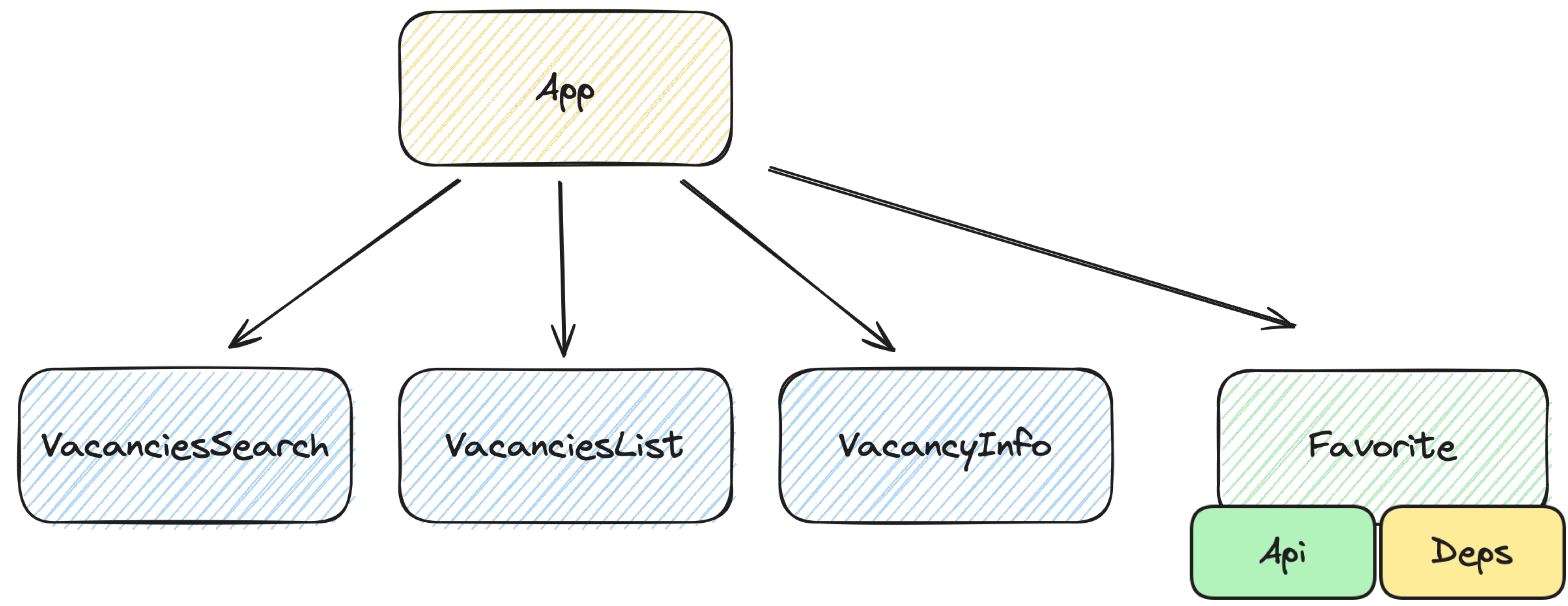
VacanciesList

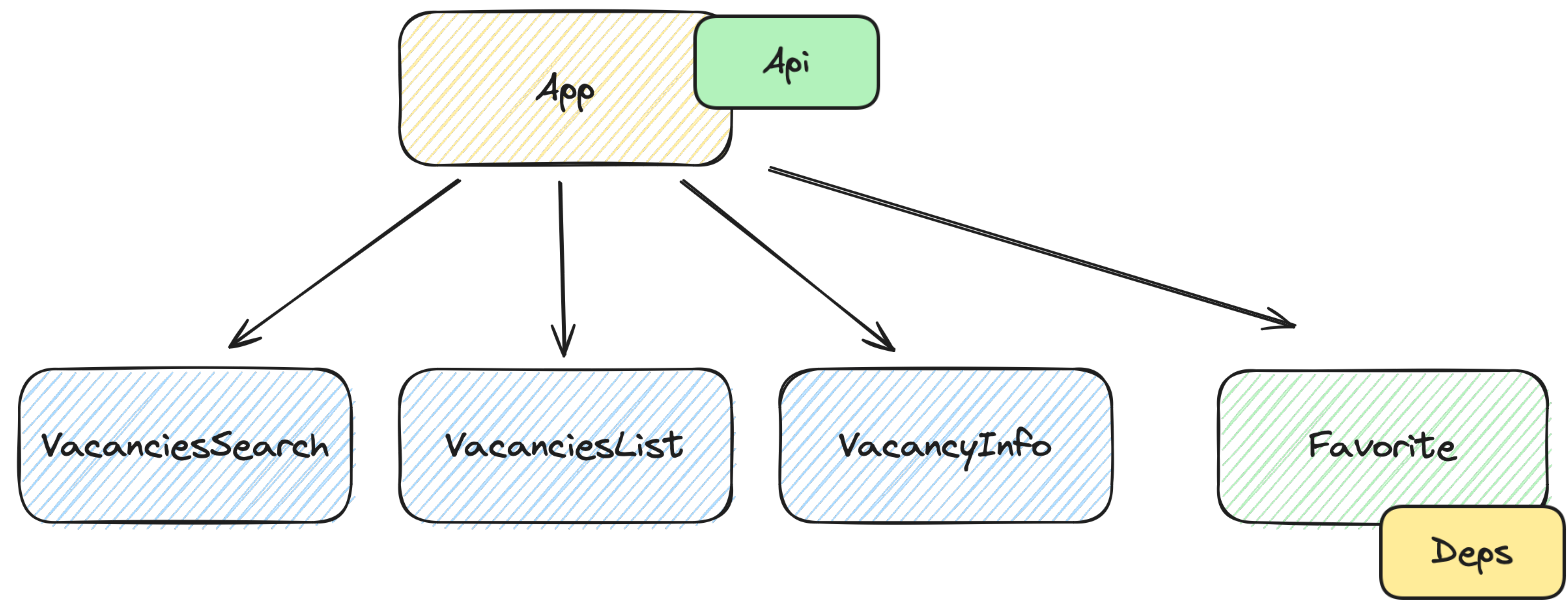
VacancyInfo

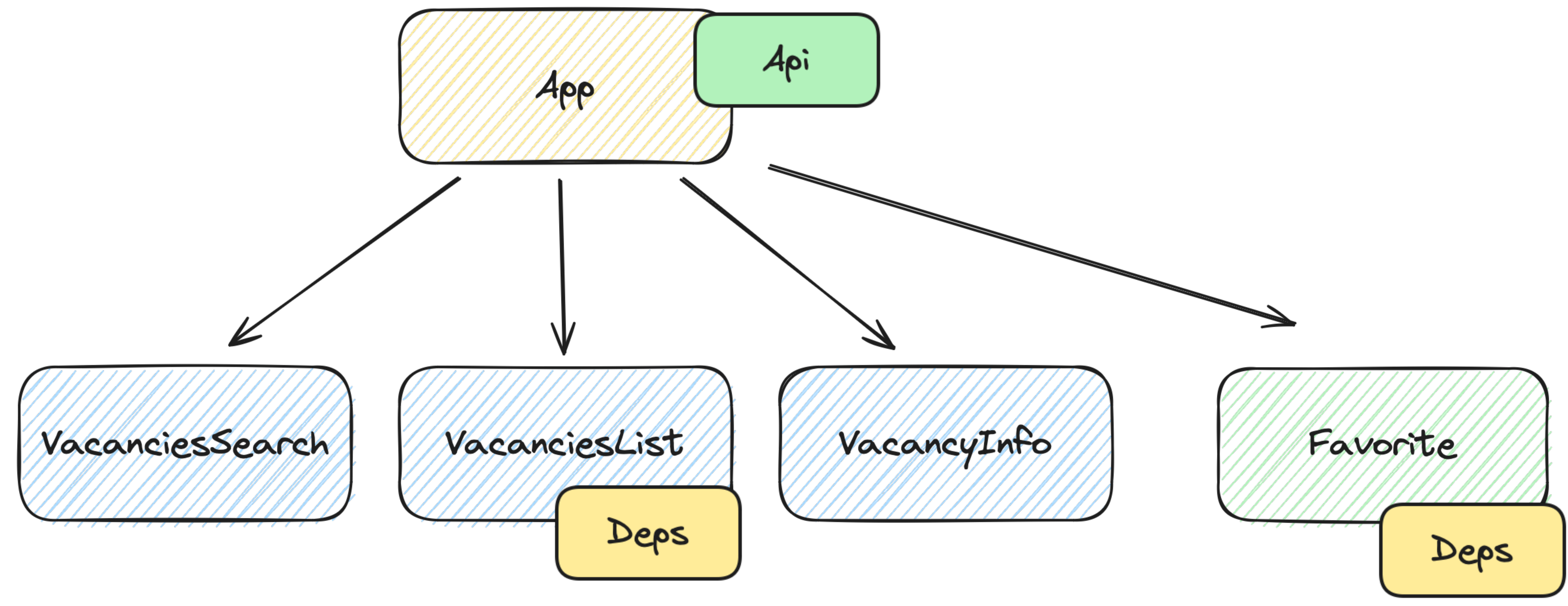

App
VacanciesSearch
VacanciesList
VacancyInfo
Favorite

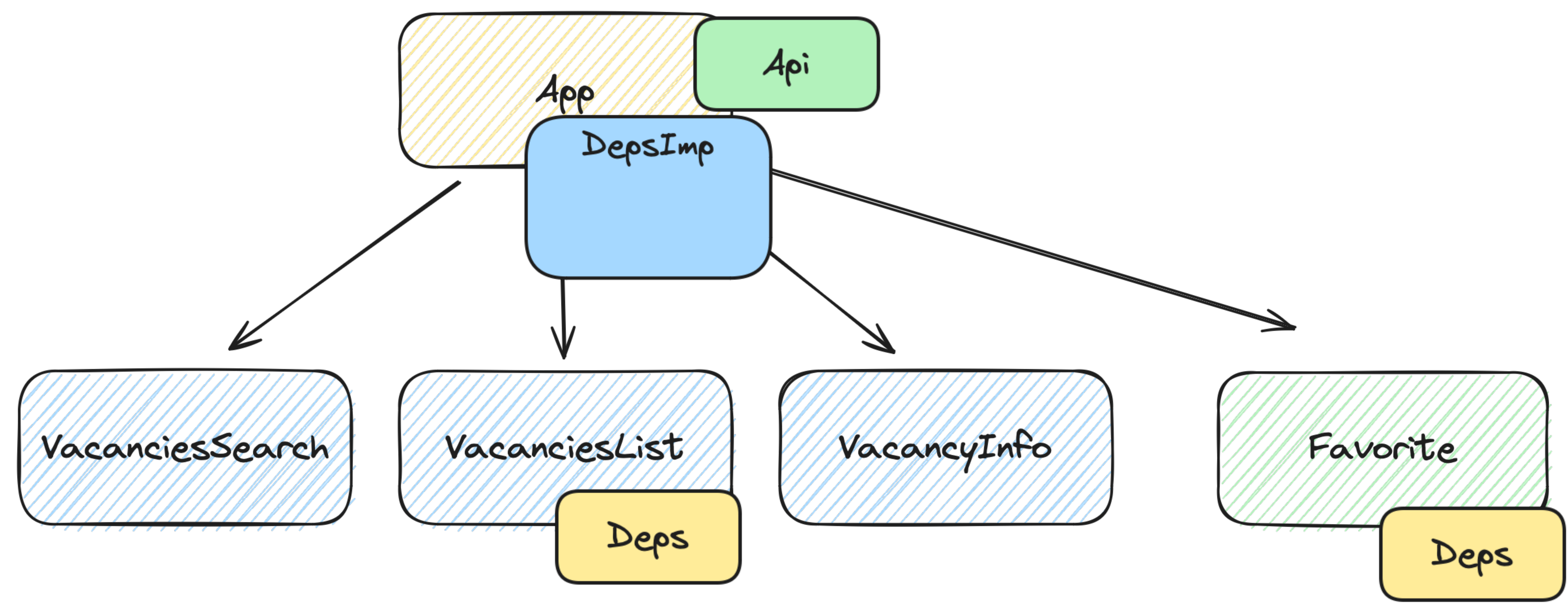

App
VacanciesSearch
VacanciesList
VacancyInfo
Favorite

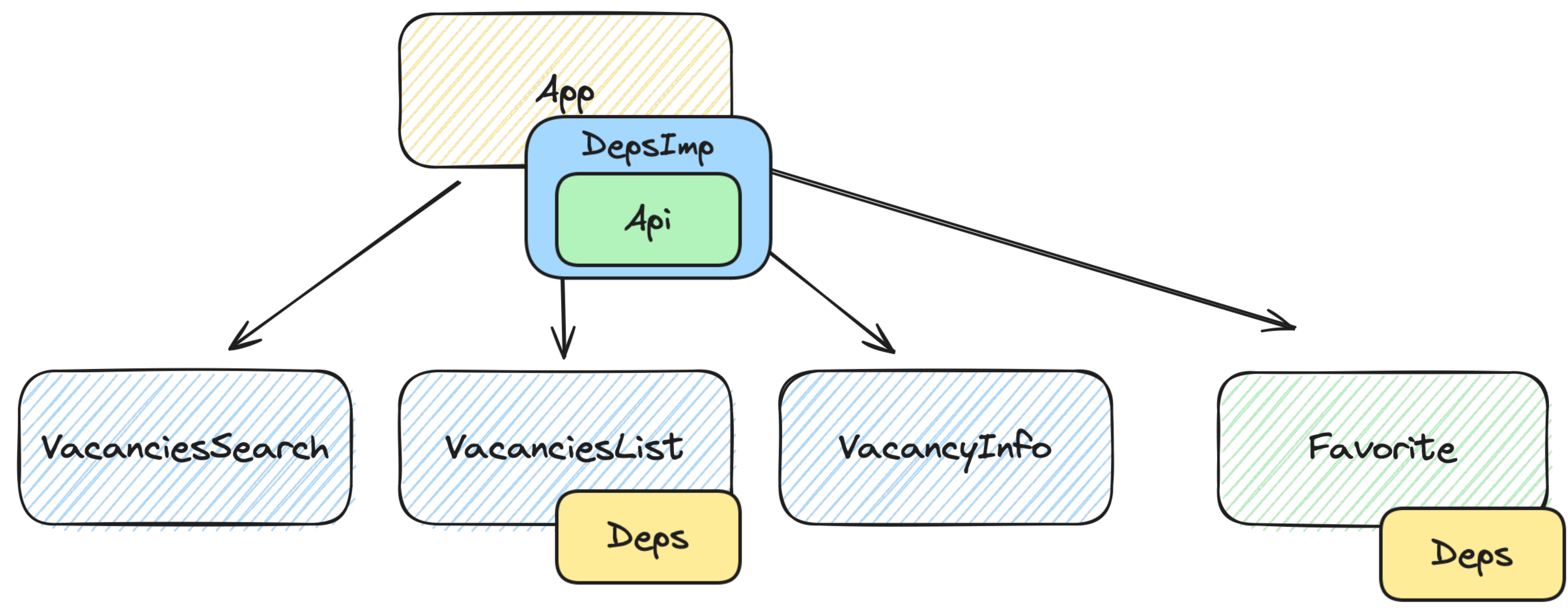

App
VacanciesSearch
VacanciesList
VacancyInfo
Favorite

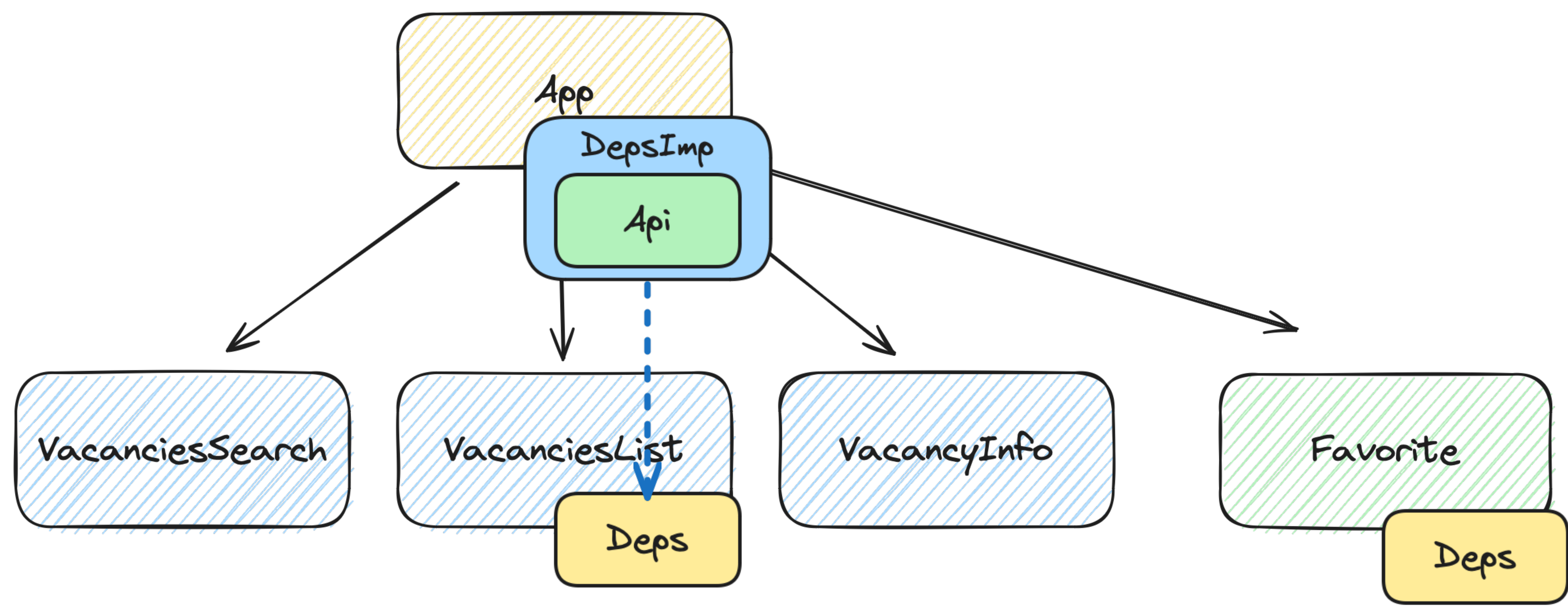

App
VacanciesSearch
VacanciesList
VacancyInfo
Favorite
Api
Deps


App
Api
VacanciesSearch
VacanciesList
VacancyInfo
Favorite
Deps


App
Api
VacanciesSearch
VacanciesList
Deps
VacancyInfo
Favorite
Deps


App
Api
DepsImp
VacanciesSearch
VacanciesList
Deps
VacancyInfo
Favorite
Deps


App
DepsImp
Api
VacanciesSearch
VacanciesList
Deps
VacancyInfo
Favorite
Deps

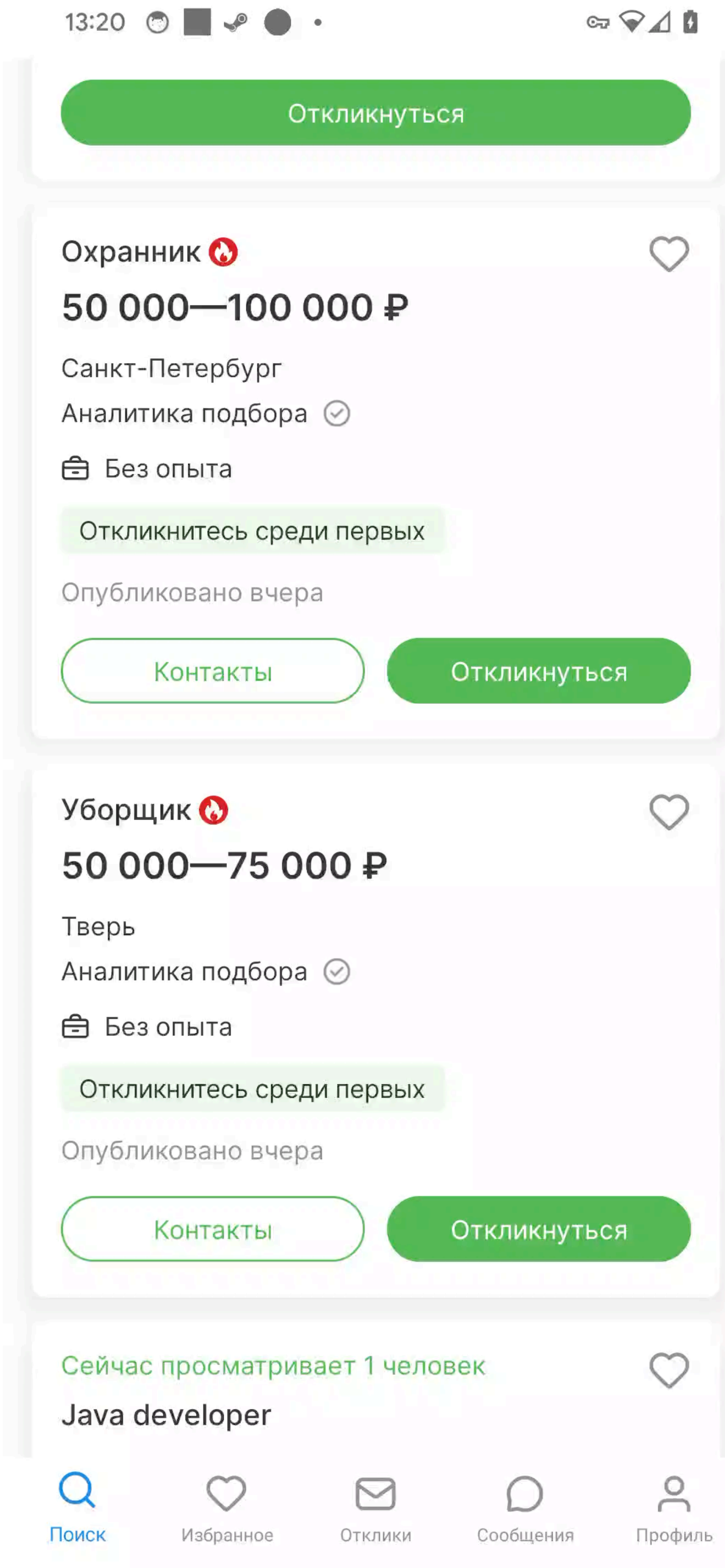

App
DepsImp
Api
VacanciesSearch
VacanciesList
Deps
VacancyInfo
Favorite
Deps

ВЛАСТЕЛИН
МОДУЛЕЙ

ВЛАСТЕЛИН
МОДУЛЕЙ

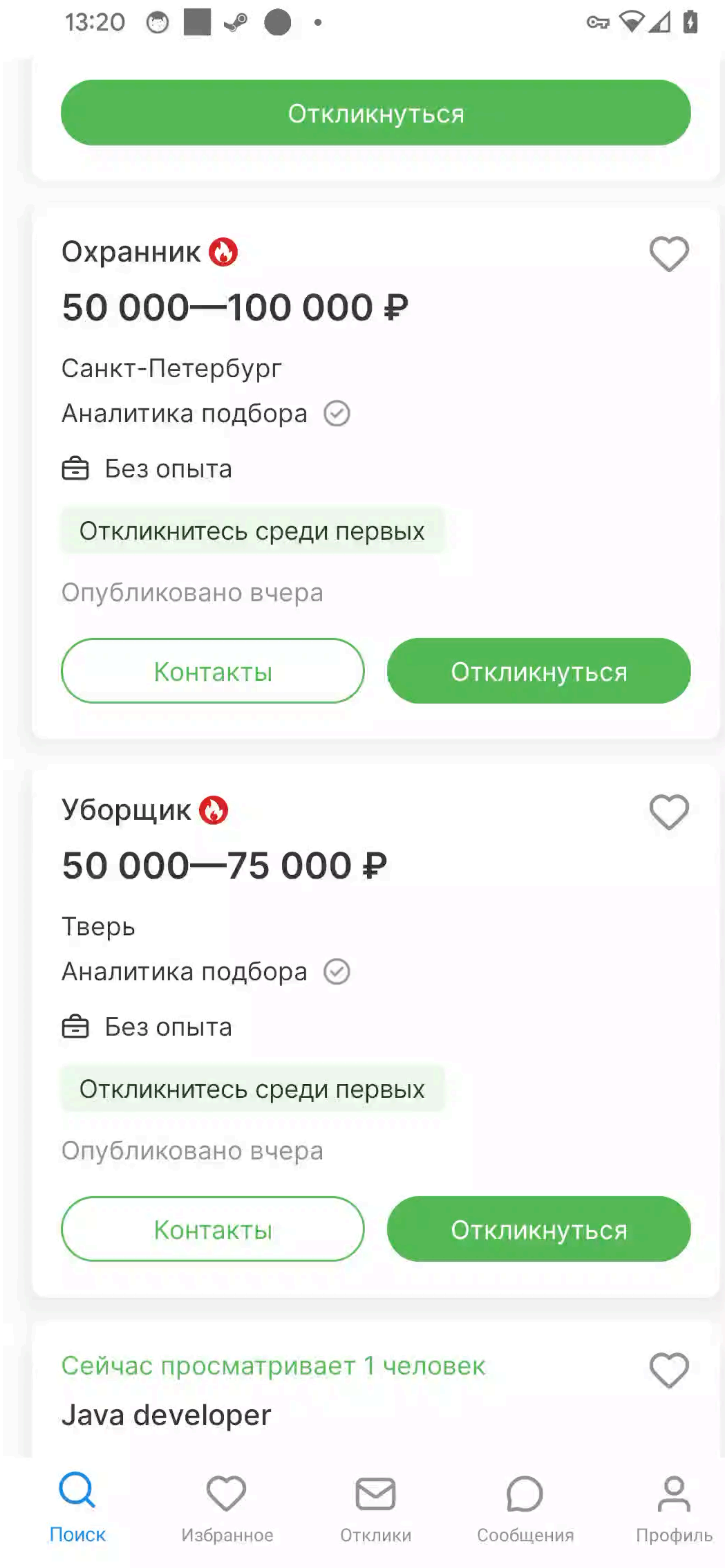
What can we share?

1. Update favorite by id
2. Observe favorite updates



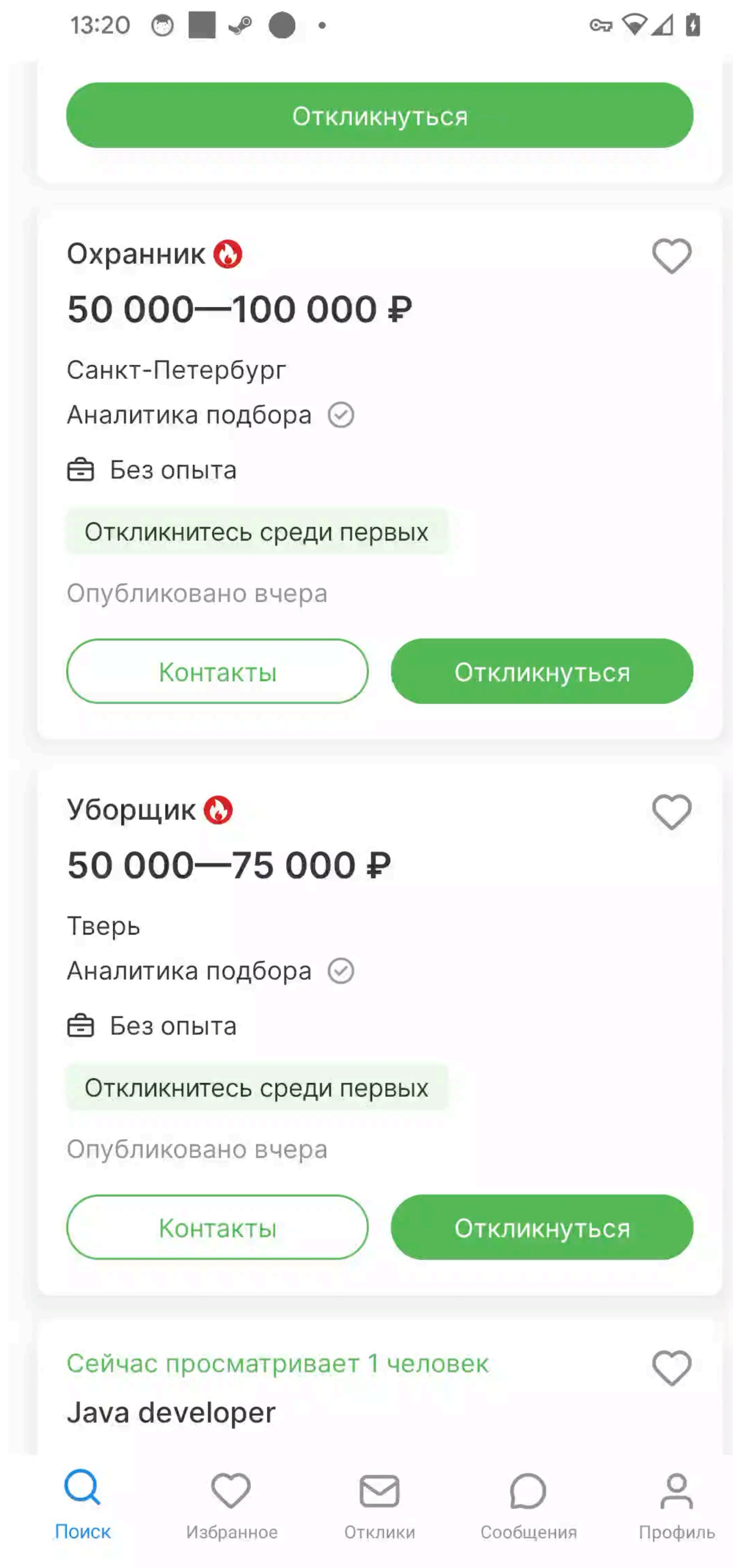
What can we share?

1. Update favorite by id
2. Observe favorite updates



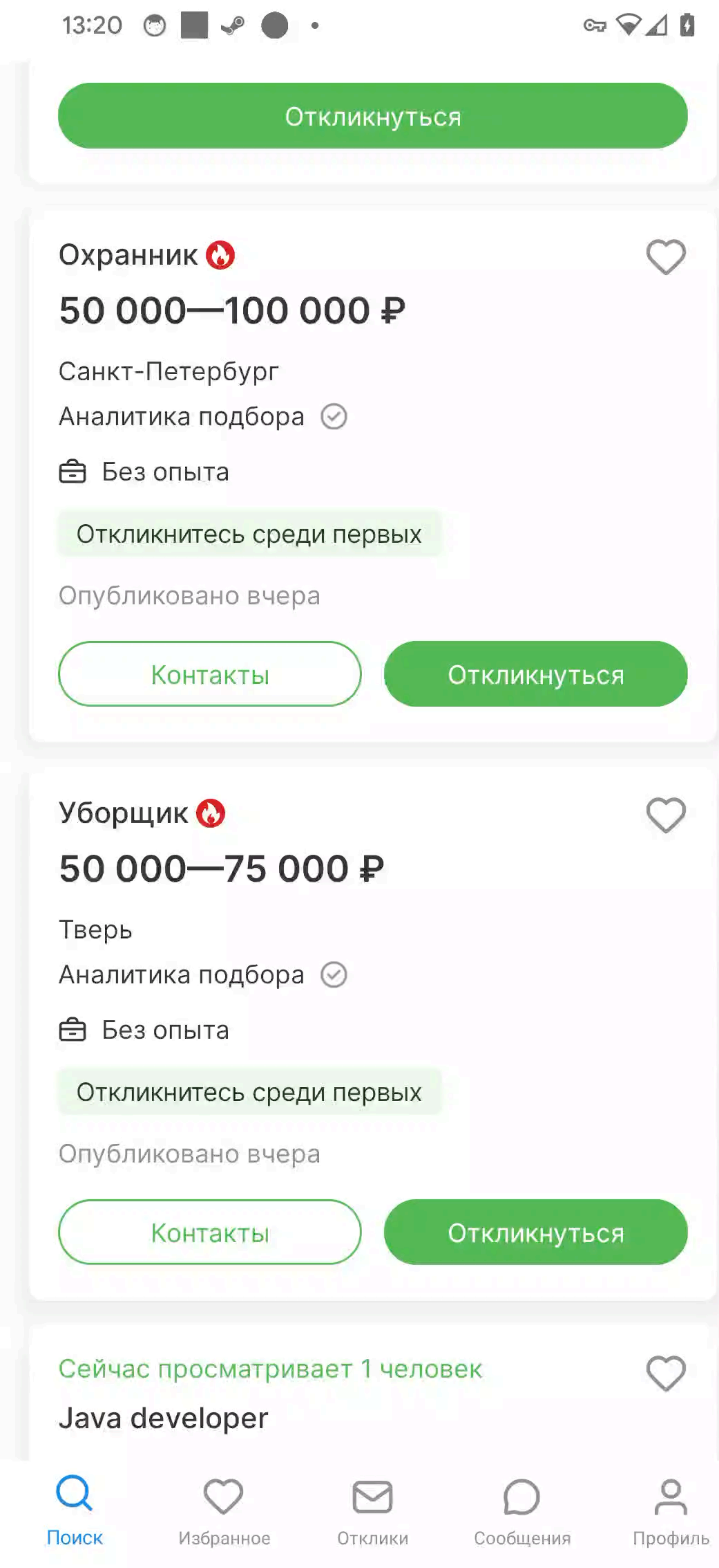
What can we share?

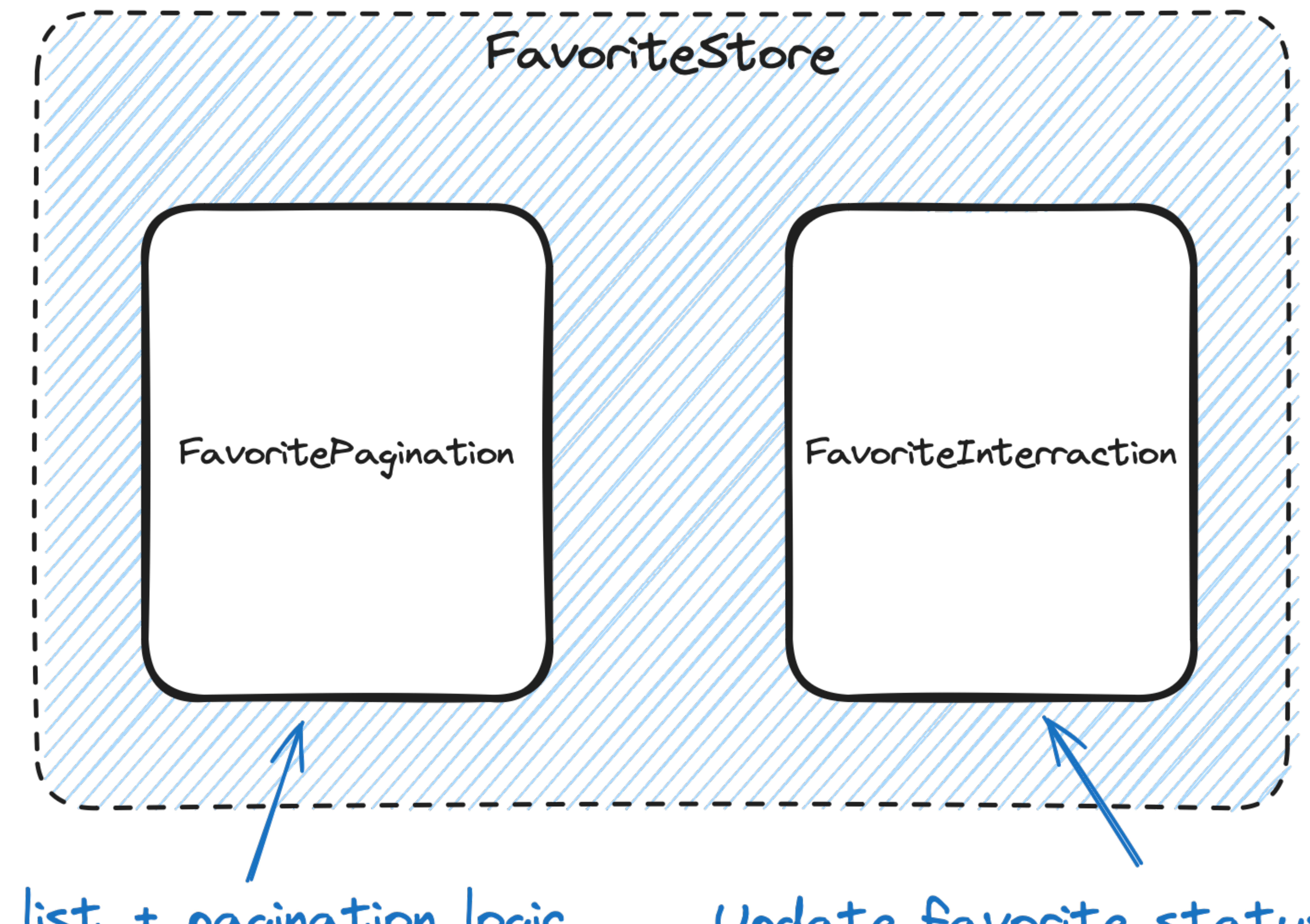
1. **Update favorite by id**
2. Observe favorite updates

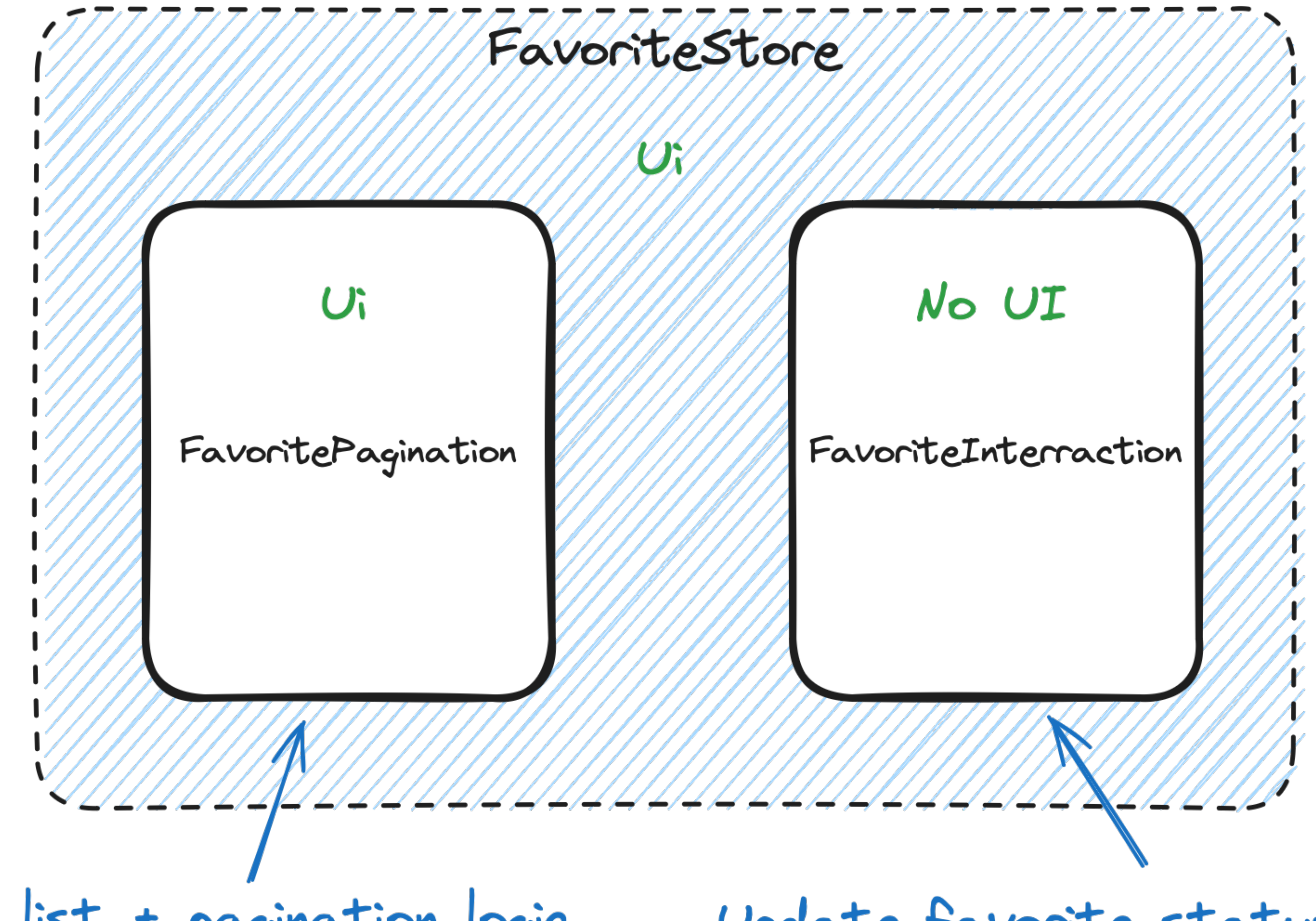


What can we share?

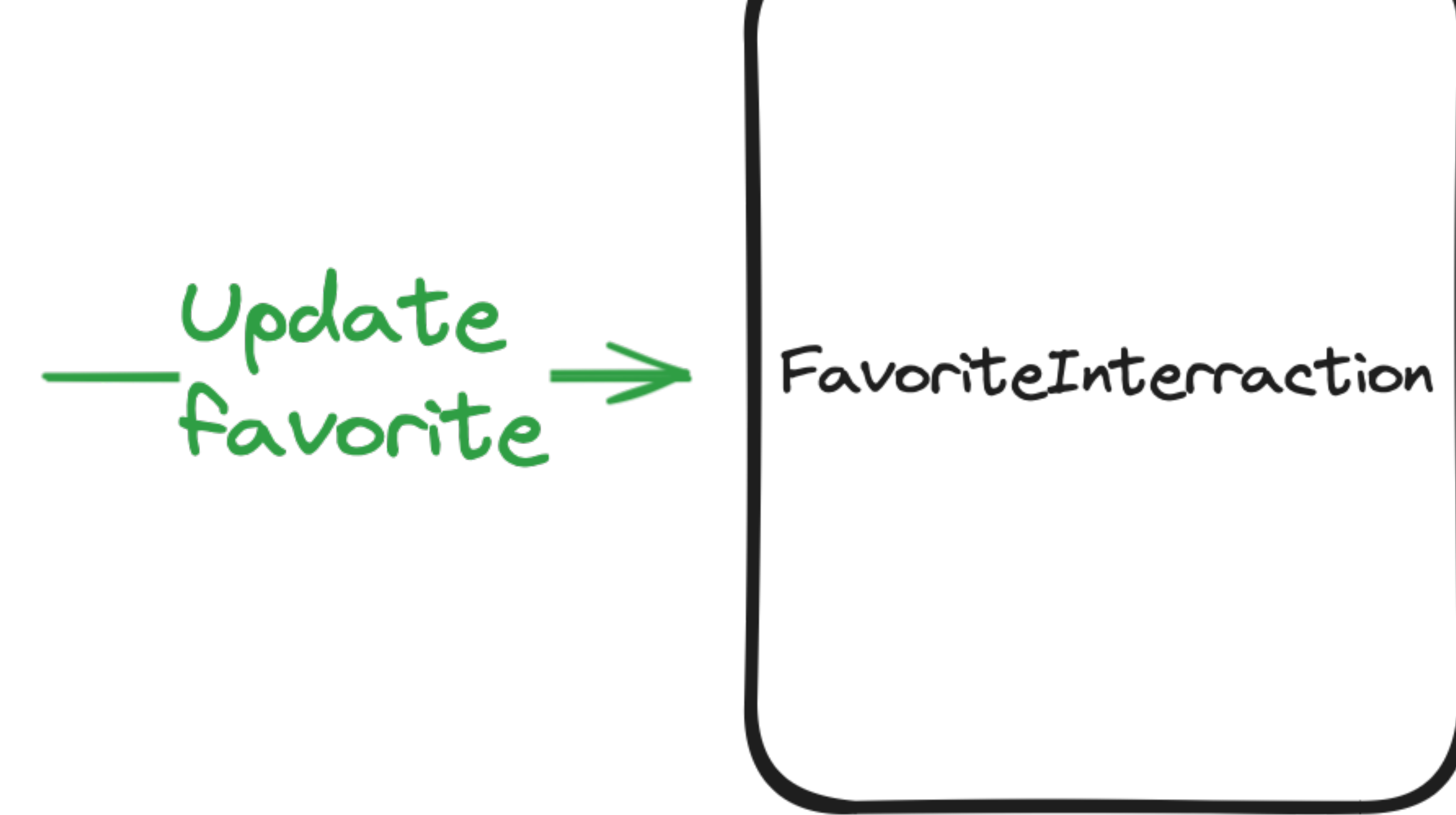
1. Update favorite by id
2. Observe favorite updates

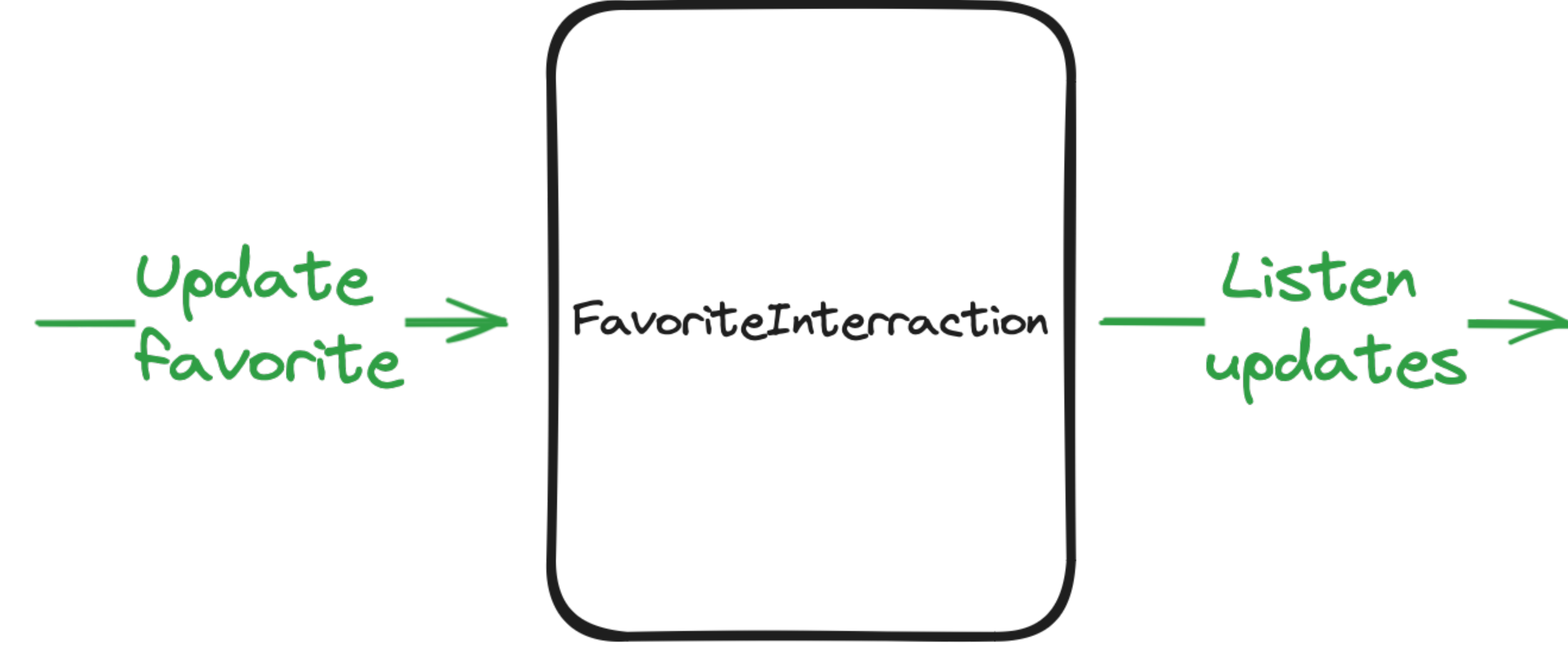



FavoriteStore
FavoritePagination
FavoriteInteraction
favorite list + pagination logic
Update favorite status logic


FavoriteStore
Ui
Ui
FavoritePagination
No UI
FavoriteInteraction
favorite list + pagination logic
Update favorite status logic

FavoriteInterraction


Update
favorite
FavoriteInterraction


Update favorite
FavoriteInterraction
Listen updates

Favorite API

```
class FavoriteApi internal constructor(
    private val favoriteInteractionStore: FavoriteInteractionStore
) {

    fun favoriteListScreen(): Screen = FavoriteScreen()

    fun updateFavorite(
        id: String,
        isFavorite: Boolean
    ) {
        favoriteInteractionStore.accept(
            FavoriteInteractionFeature.Msg.Outer.UpdateFavorite(id, isFavorite)
        )
    }

    fun observeFavoriteUpdates(): Flow<FavoriteUpdate> =
        favoriteInteractionStore.effects
            .mapNotNull { eff ->
                when (eff) {
                    is FavoriteInteractionFeature.Eff.Outer.ItemUpdate.Started ->
```

Favorite API

```
class FavoriteApi internal constructor(
    private val favoriteInteractionStore: FavoriteInteractionStore
) {

    fun favoriteListScreen(): Screen = FavoriteScreen()

    fun updateFavorite(
        id: String,
        isFavorite: Boolean
    ) {
        favoriteInteractionStore.accept(
            FavoriteInteractionFeature.Msg.Outer.UpdateFavorite(id, isFavorite)
        )
    }

    fun observeFavoriteUpdates(): Flow<FavoriteUpdate> =
        favoriteInteractionStore.effects
            .mapNotNull { eff ->
                when (eff) {
                    is FavoriteInteractionFeature.Eff.Outer.ItemUpdate.Started ->
```

Favorite API

```
class FavoriteApi internal constructor(
    private val favoriteInteractionStore: FavoriteInteractionStore
) {

    fun favoriteListScreen(): Screen = FavoriteScreen()

    fun updateFavorite(
        id: String,
        isFavorite: Boolean
    ) {
        favoriteInteractionStore.accept(
            FavoriteInteractionFeature.Msg.Outer.UpdateFavorite(id, isFavorite)
        )
    }

    fun observeFavoriteUpdates(): Flow<FavoriteUpdate> =
        favoriteInteractionStore.effects
            .mapNotNull { eff ->
                when (eff) {
                    is FavoriteInteractionFeature.Eff.Outer.ItemUpdate.Started ->
```

Favorite API

```
class FavoriteApi internal constructor(
    private val favoriteInteractionStore: FavoriteInteractionStore
) {

    fun favoriteListScreen(): Screen = FavoriteScreen()

    fun updateFavorite(
        id: String,
        isFavorite: Boolean
    ) {
        favoriteInteractionStore.accept(
            FavoriteInteractionFeature.Msg.Outer.UpdateFavorite(id, isFavorite)
        )
    }

    fun observeFavoriteUpdates(): Flow<FavoriteUpdate> =
        favoriteInteractionStore.effects
            .mapNotNull { eff ->
                when (eff) {
                    is FavoriteInteractionFeature.Eff.Outer.ItemUpdate.Started ->
```

Favorite API

```
data class FavoriteUpdate(
    val id: String,
    val isFavorite: Boolean
)
```

Favorite updates in API

```
class FavoriteApi internal constructor(
    private val favoriteStore: FavoriteStore
) {

    fun observeFavoriteUpdates(): Flow<FavoriteUpdate> =
        favoriteStore.effects
            .mapNotNull { eff ->
                when (eff) {
                    is FavoriteFeature.Eff.Outer.ItemUpdate.Started ->
                         eff.item
                    is FavoriteFeature.Eff.Outer.ItemUpdate.Error ->
                         eff.item.run { copy(isFavorite = !isFavorite) }
                    else -> null
                }
            }

}
```

Favorite updates in API

```
class FavoriteApi internal constructor(
    private val favoriteStore: FavoriteStore
) {

    fun observeFavoriteUpdates(): Flow<FavoriteUpdate> =
        favoriteStore.effects
            .mapNotNull { eff ->
                when (eff) {
                    is FavoriteFeature.Eff.Outer.ItemUpdate.Started ->
                        eff.item
                    is FavoriteFeature.Eff.Outer.ItemUpdate.Error ->
                        eff.item.run { copy(isFavorite = !isFavorite) }
                    else -> null
                }
            }

}
```

Favorite updates in API

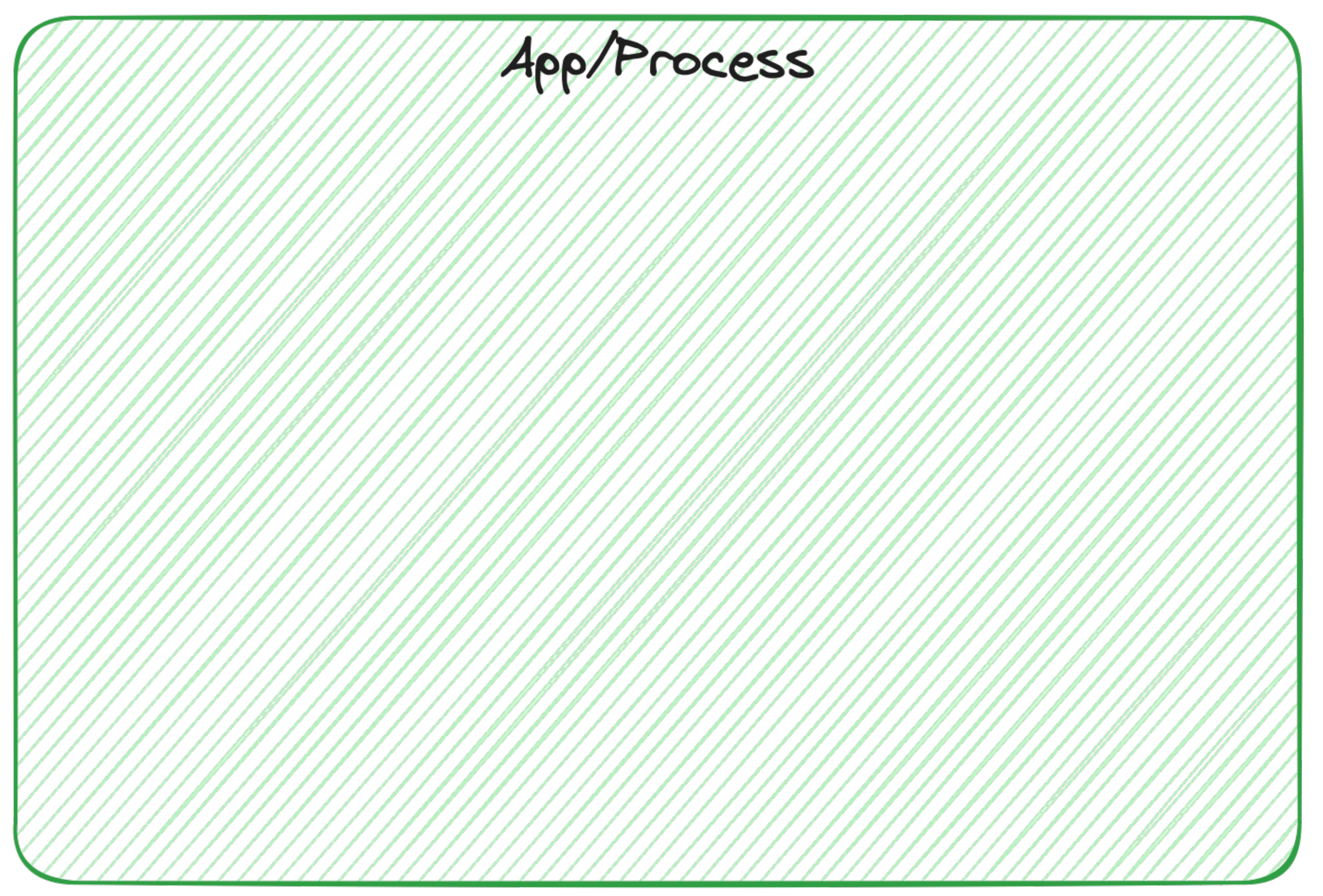
```
class FavoriteApi internal constructor(
    private val favoriteStore: FavoriteStore
) {

    fun observeFavoriteUpdates(): Flow<FavoriteUpdate> =
        favoriteStore.effects
            .mapNotNull { eff ->
                when (eff) {
                    is FavoriteFeature.Eff.Outer.ItemUpdate.Started ->
                         eff.item
                    is FavoriteFeature.Eff.Outer.ItemUpdate.Error ->
                         eff.item.run { copy(isFavorite = !isFavorite) }
                    else -> null
                }
            }

}
```

FavoriteApi

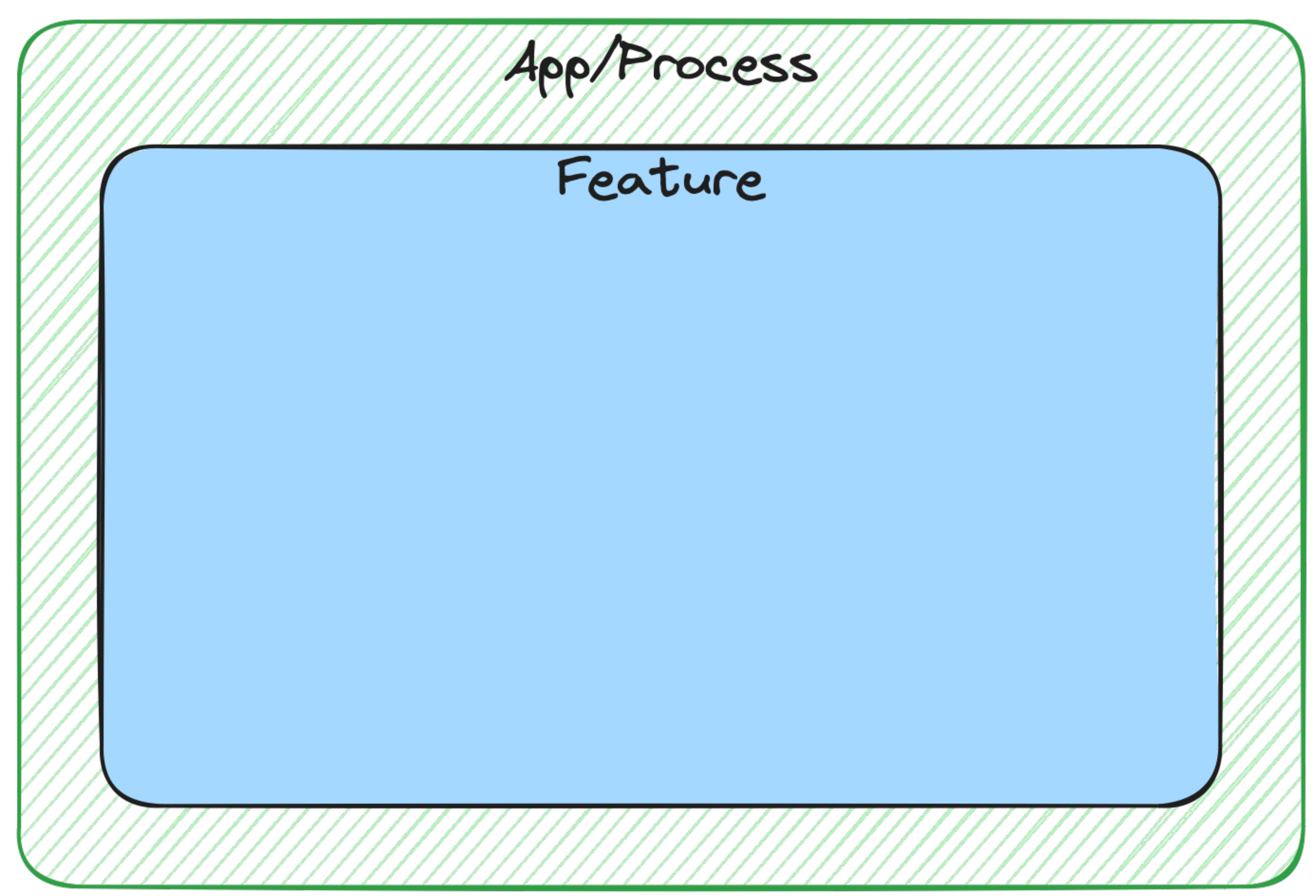
Done

Store
Lifecycle

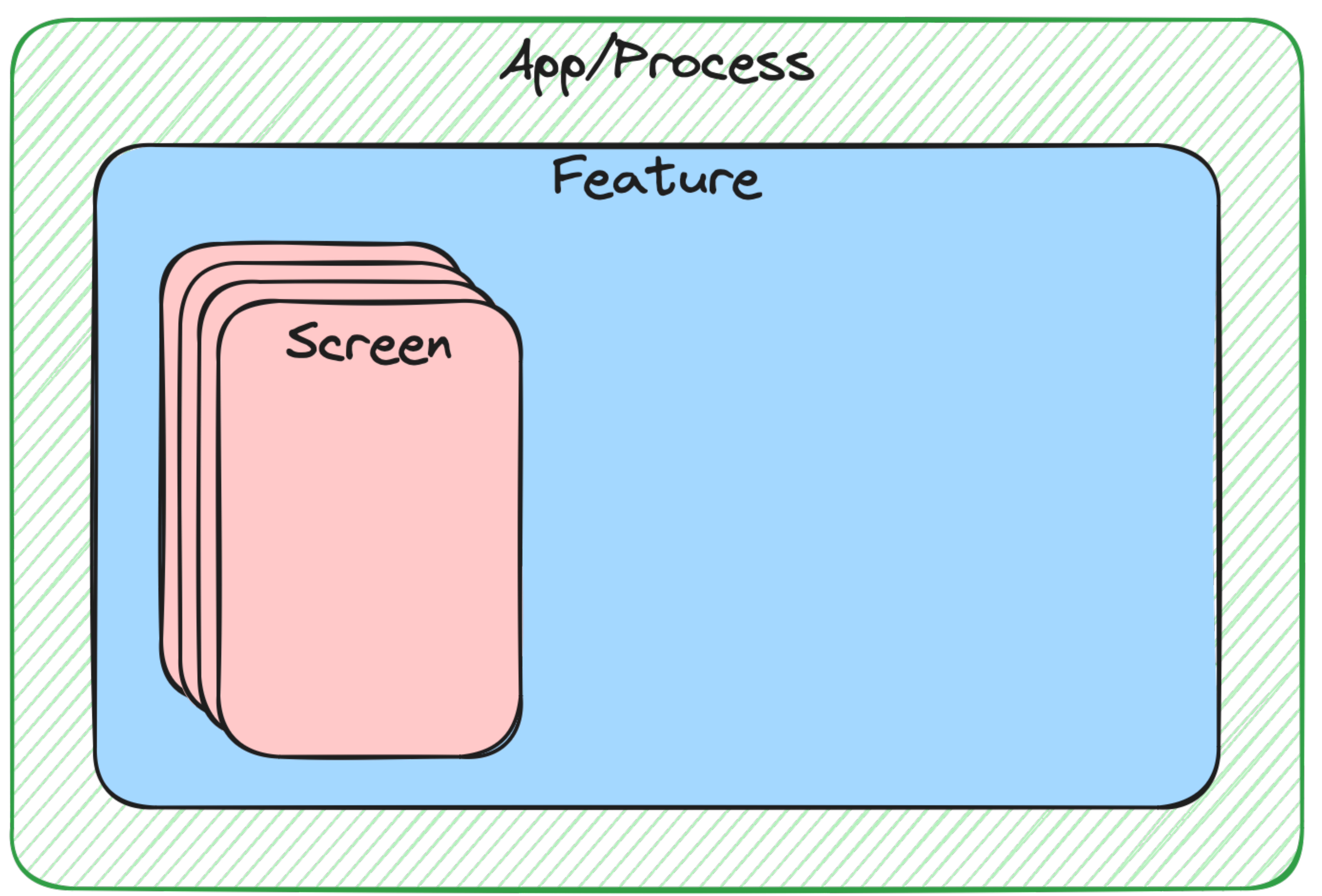
Lifecycle



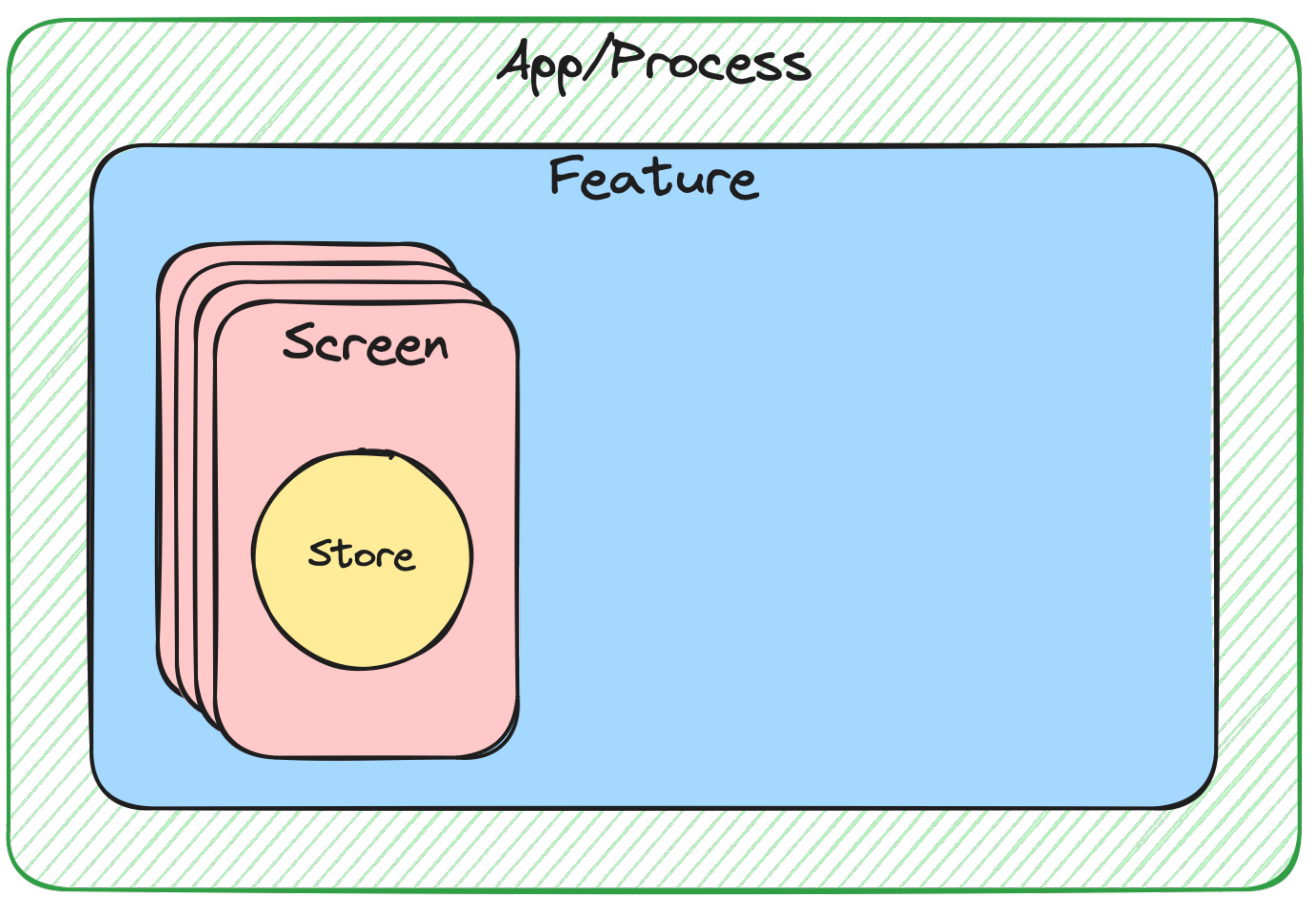
Lifecycle



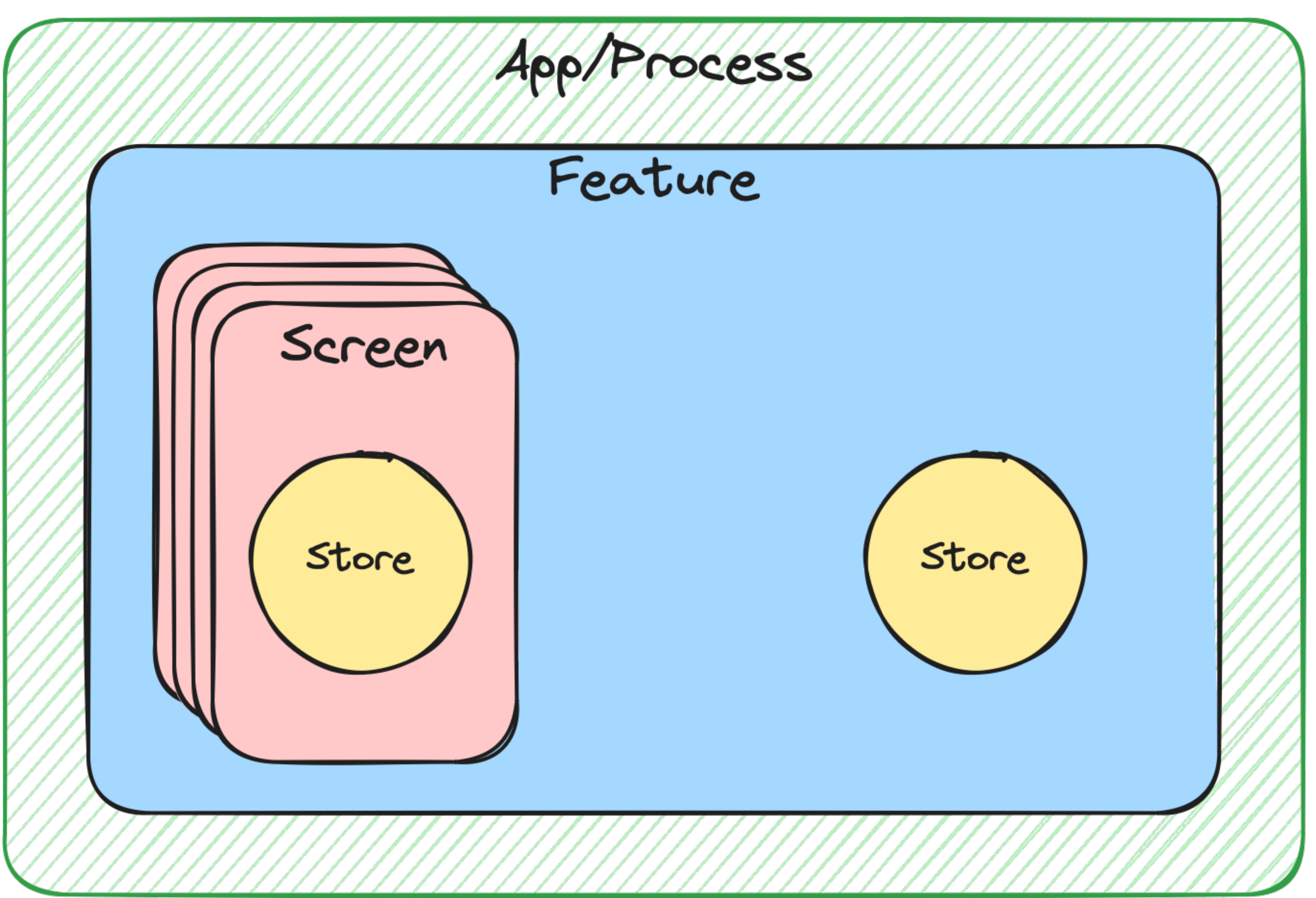
Lifecycle



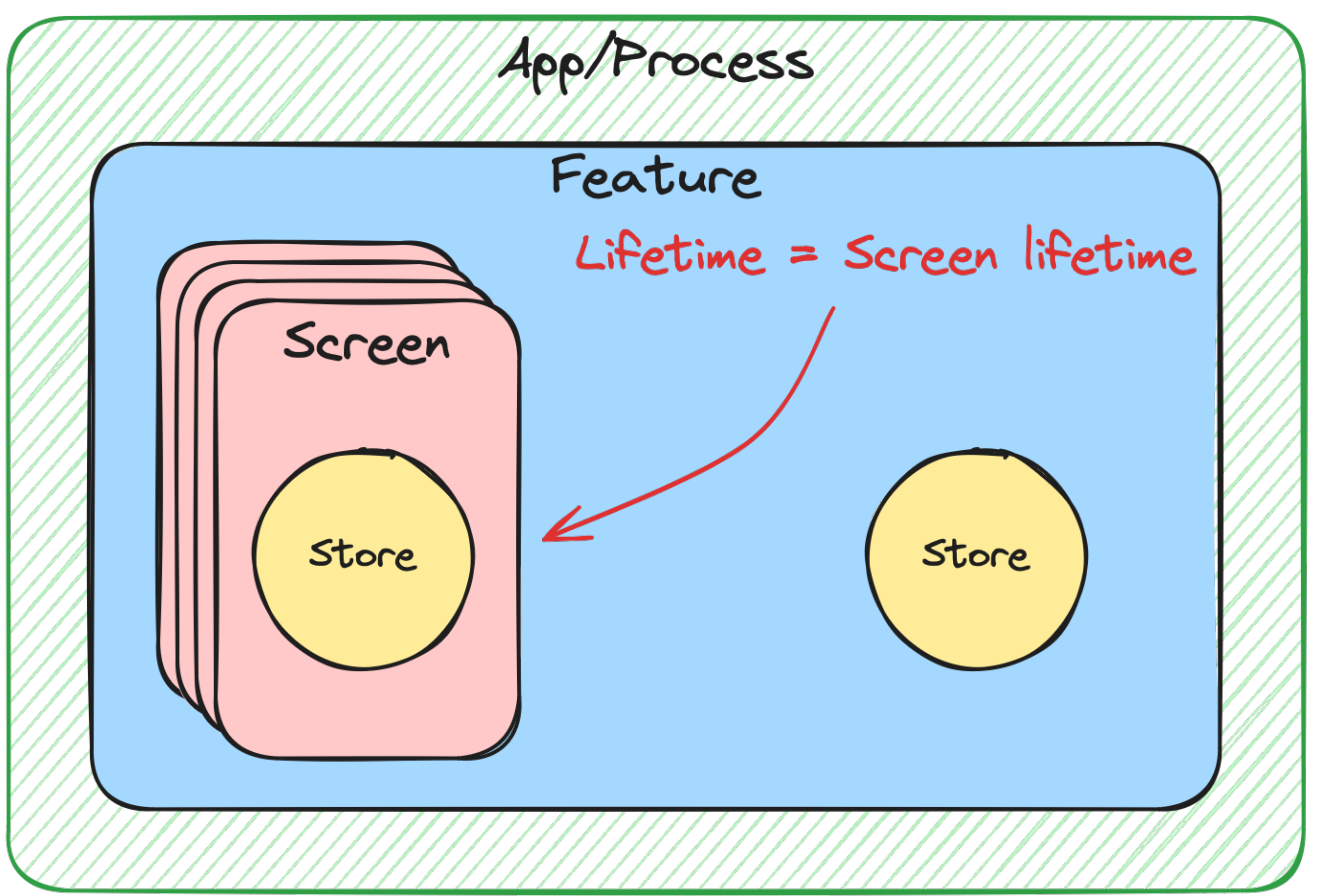
Lifecycle



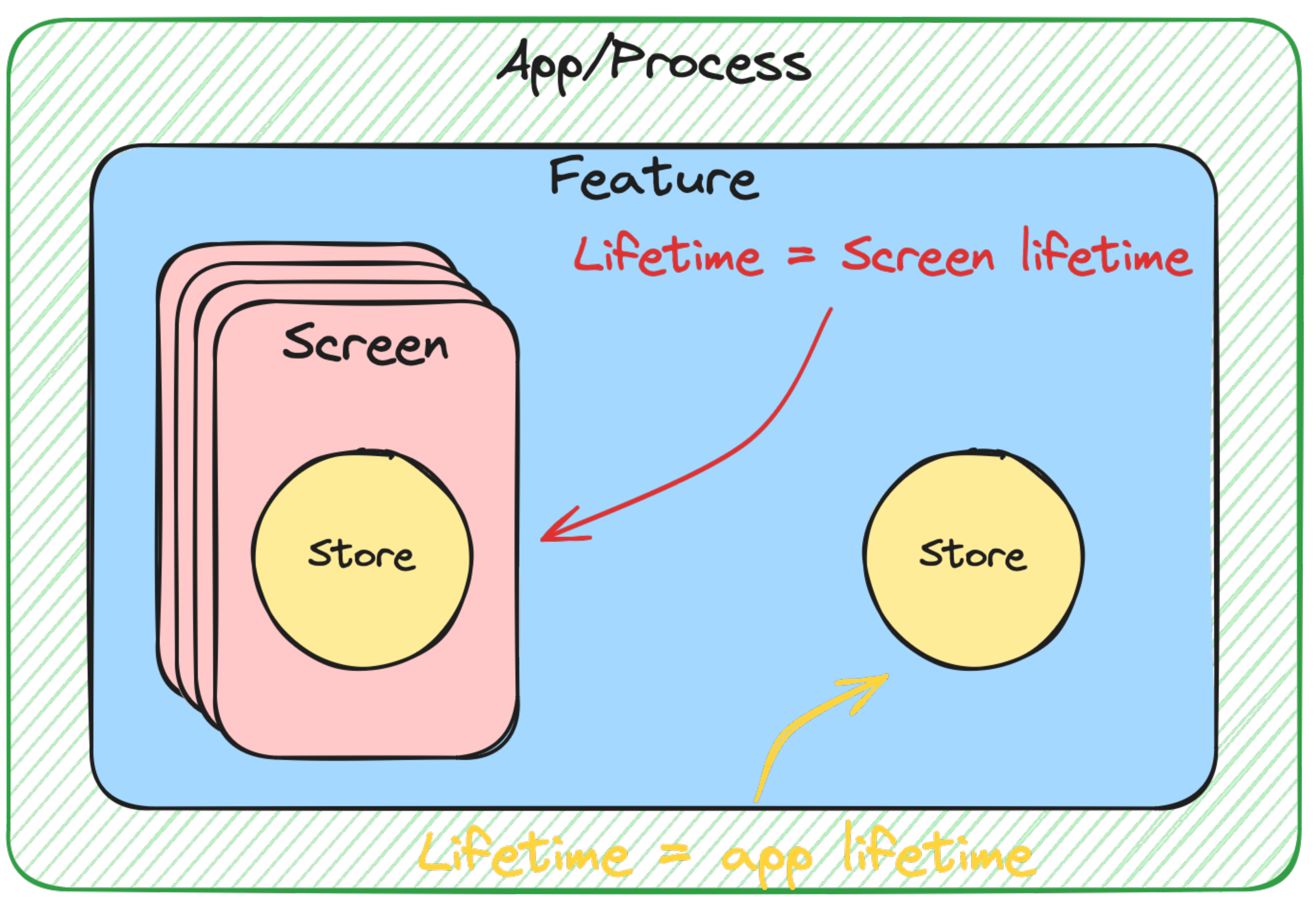
Lifecycle



Lifecycle



Lifecycle



FavoriteInteraction - Store without UI!

Correct cancelation

```
internal class FavoriteAggregatorStore(
    private val paginationStore: FavoritePaginationStore,
    private val interactionStore: FavoriteInteractionStore,
) : AggregatorStore<Msg, State, Eff>(
    name = "FavoriteAggregatorStore"
) {
    ...

    override fun close() {
        paginationStore.close()
        super.close()
    }

}
```

Correct cancelation

```
internal class FavoriteAggregatorStore(
    private val paginationStore: FavoritePaginationStore,
    private val interactionStore: FavoriteInteractionStore,
) : AggregatorStore<Msg, State, Eff>(
    name = "FavoriteAggregatorStore"
) {
    ...

    override fun close() {
        paginationStore.close()
        super.close()
    }

}
```

Correct cancelation

```
internal class FavoriteAggregatorStore(
    private val paginationStore: FavoritePaginationStore,
    private val interactionStore: FavoriteInteractionStore,
) : AggregatorStore<Msg, State, Eff>(
    name = "FavoriteAggregatorStore"
) {
    ...

    override fun close() {
        paginationStore.close()
        super.close()
    }

}
```

Correct cancelation

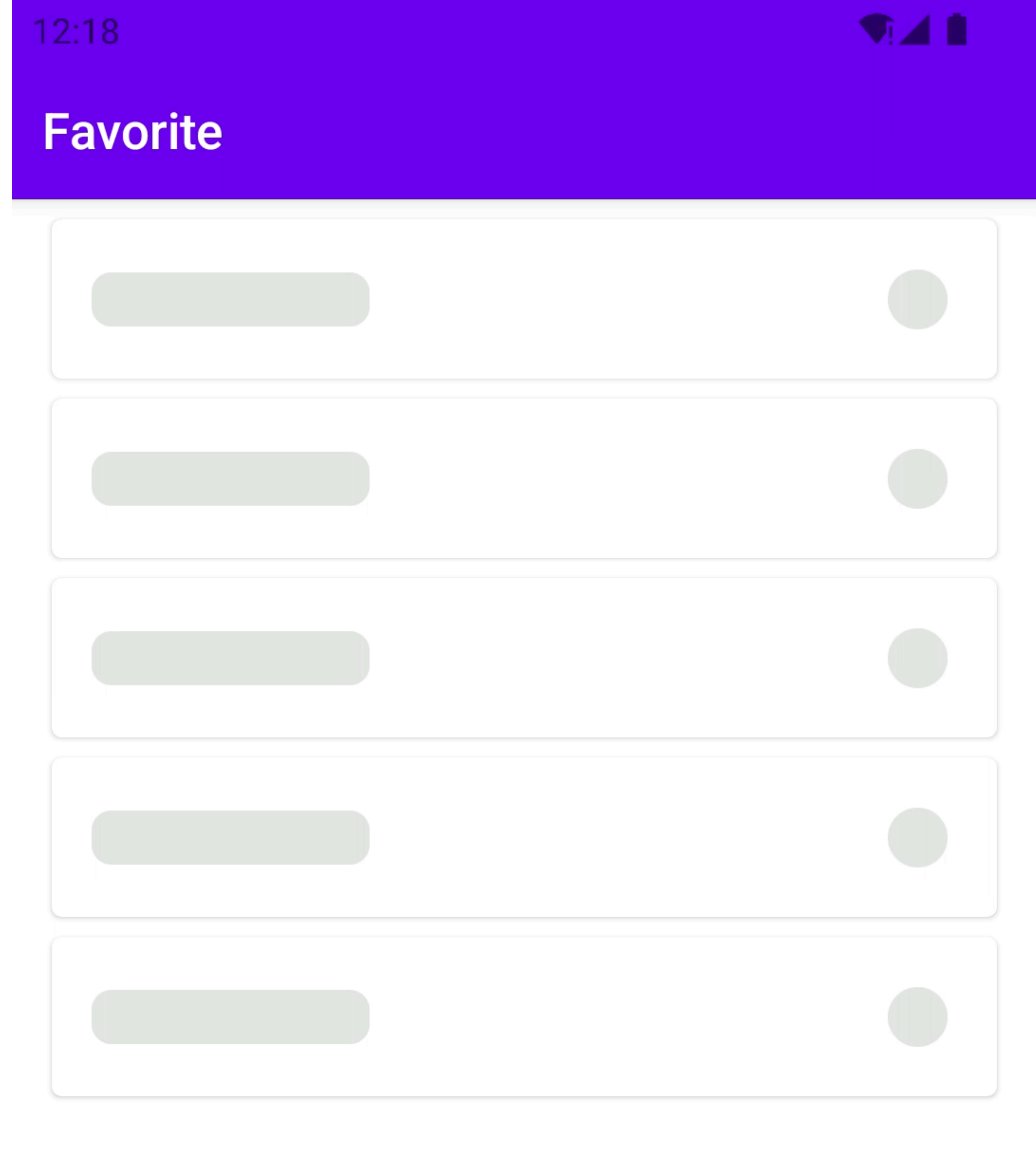
```
internal class FavoriteAggregatorStore(
    private val paginationStore: FavoritePaginationStore,
    private val interactionStore: FavoriteInteractionStore,
) : AggregatorStore<Msg, State, Eff>(
    name = "FavoriteAggregatorStore"
) {
    ...

    override fun close() {
        paginationStore.close()
        super.close()
    }

}
```

We don't close interaction store!

Are we done
With favorite?

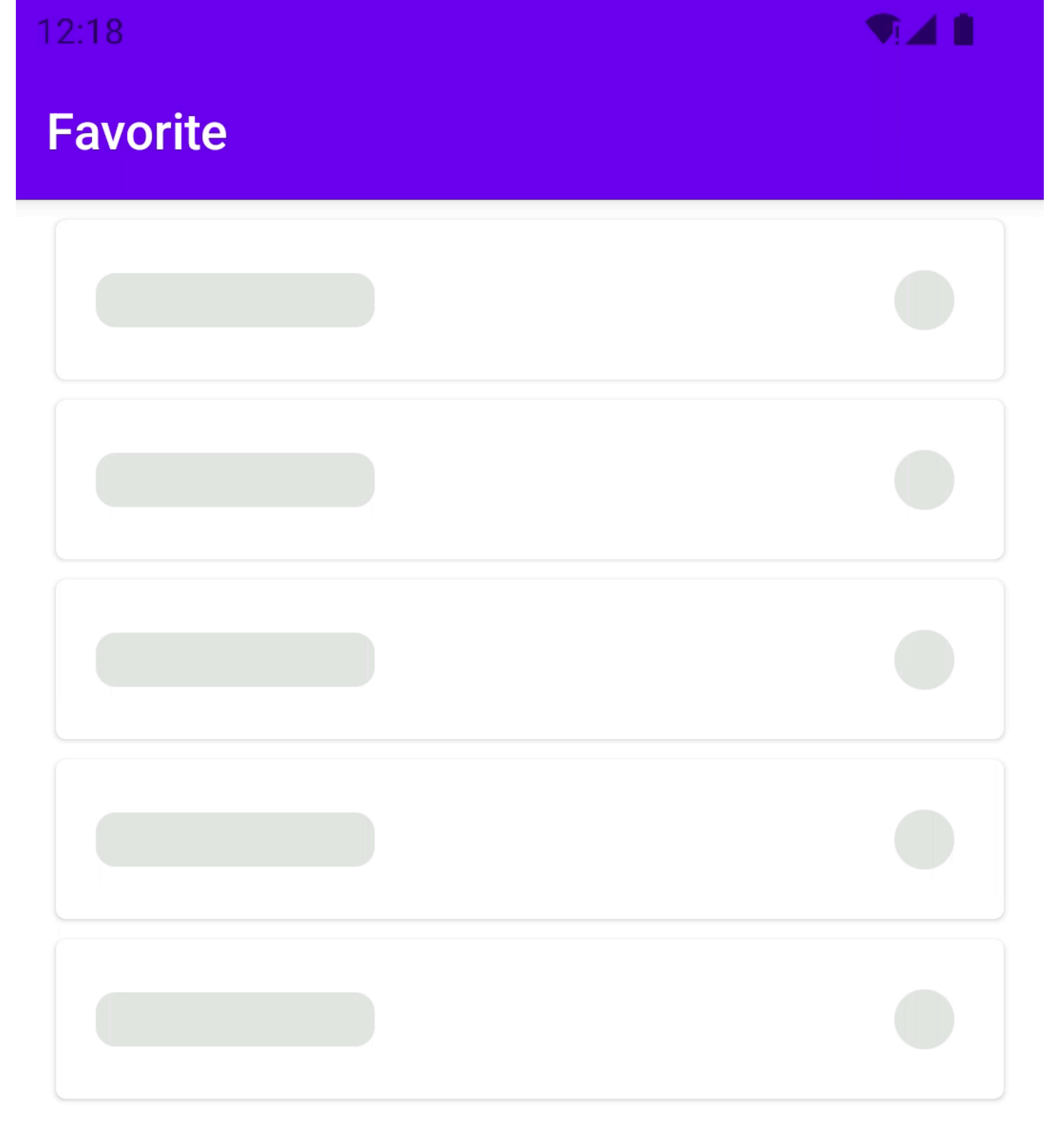
Test process death (PD)

1. Start app
2. Wait success loading
3. Go to home screen
4. Kill app process
5. Return to the app
6. There is loading again!



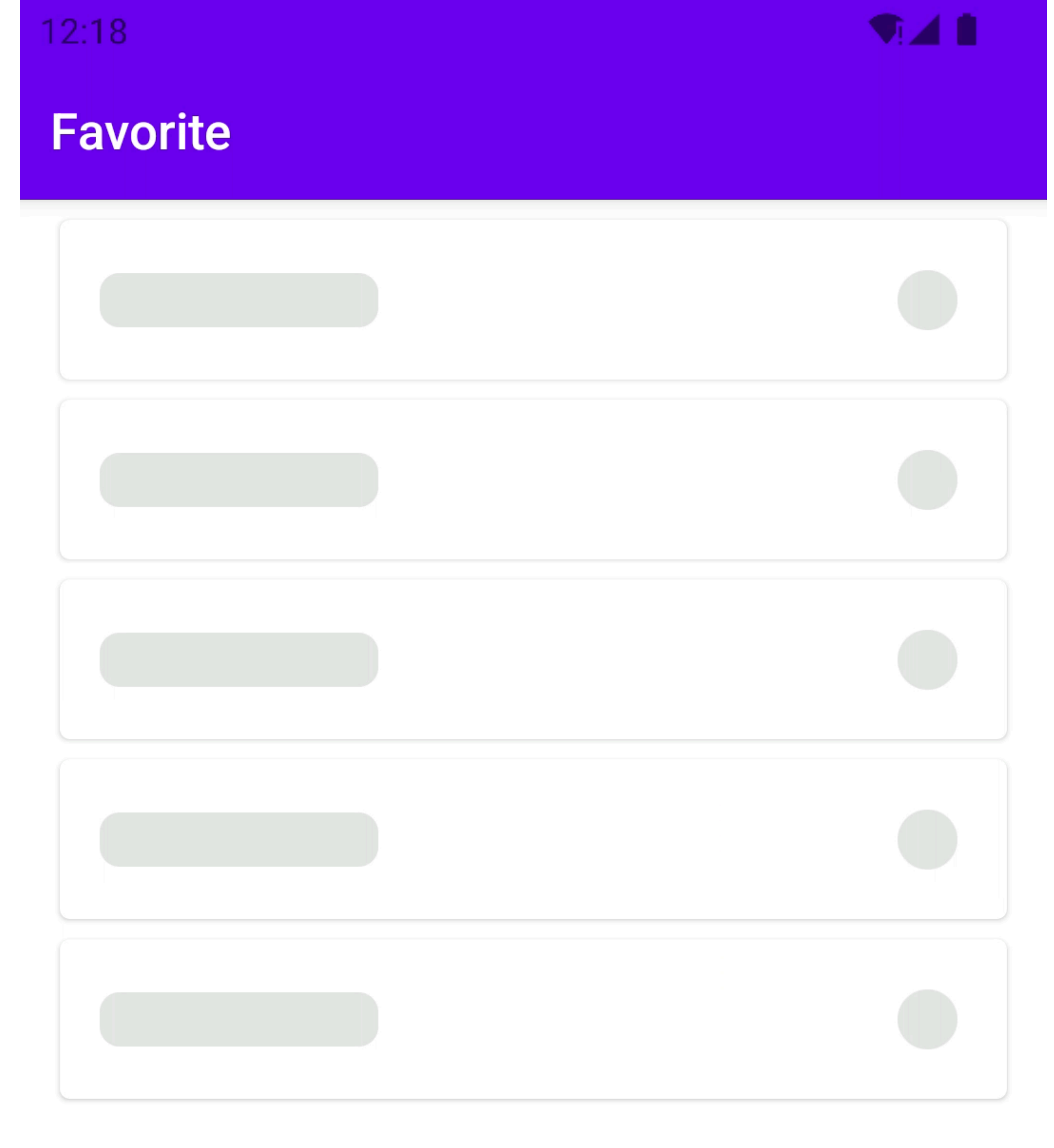
Test process death (PD)

1. **Start app**
2. Wait success loading
3. Go to home screen
4. Kill app process
5. Return to the app
6. There is loading again!



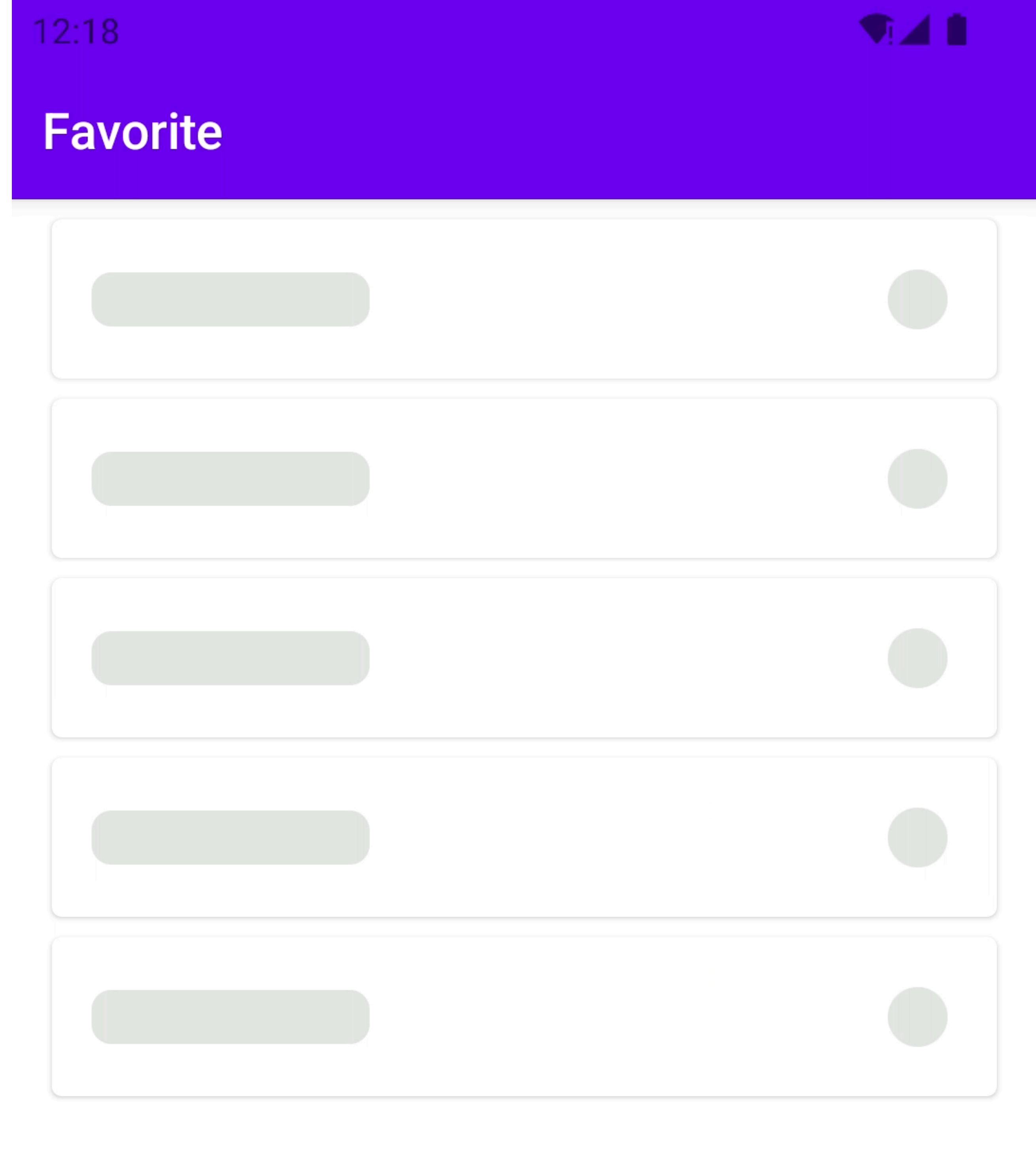
Test process death (PD)

1. **Start app**
2. **Wait success loading**
3. Go to home screen
4. Kill app process
5. Return to the app
6. There is loading again!



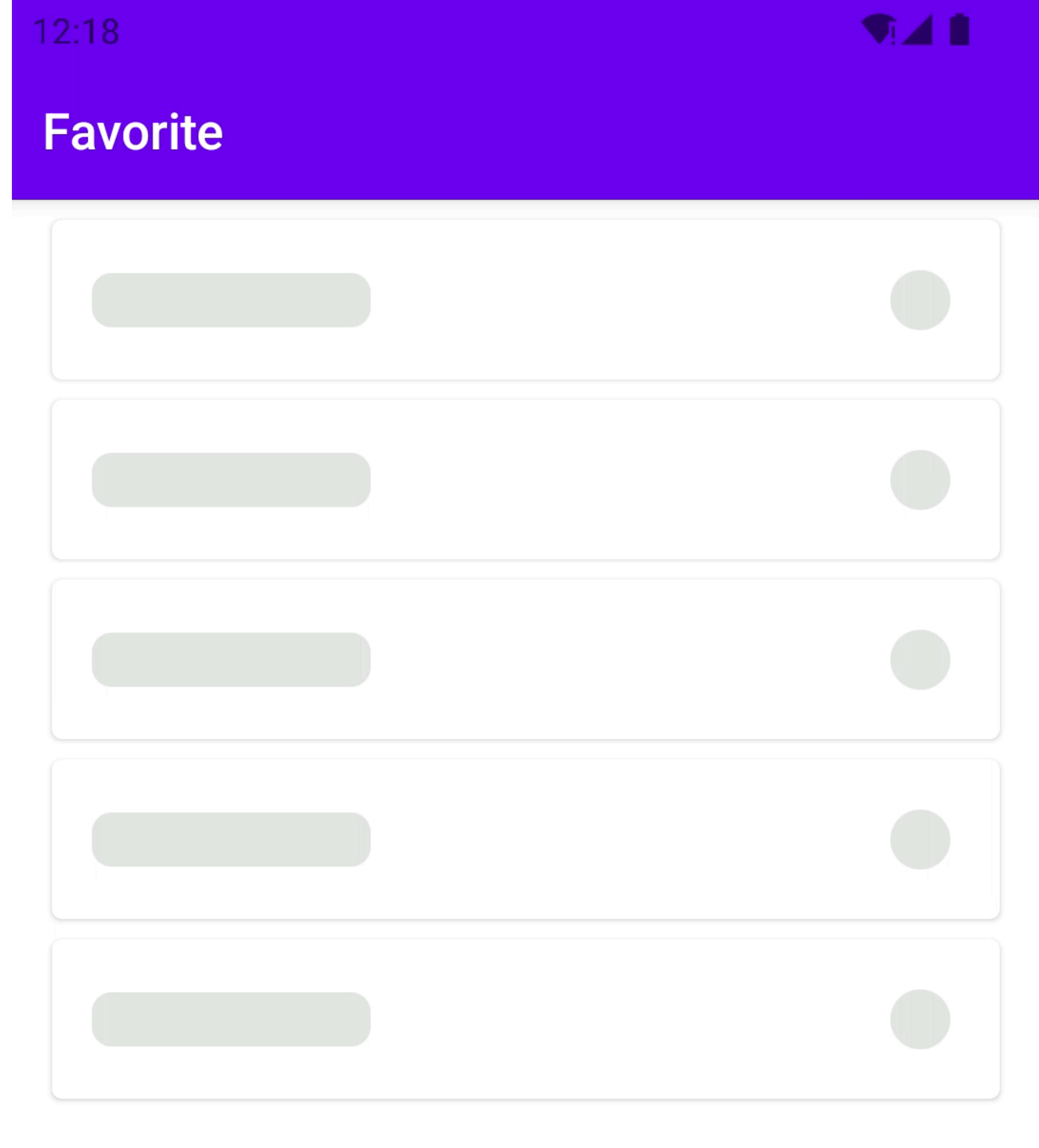
Test process death (PD)

1. Start app
2. Wait success loading
3. Go to home screen
4. Kill app process
5. Return to the app
6. There is loading again!



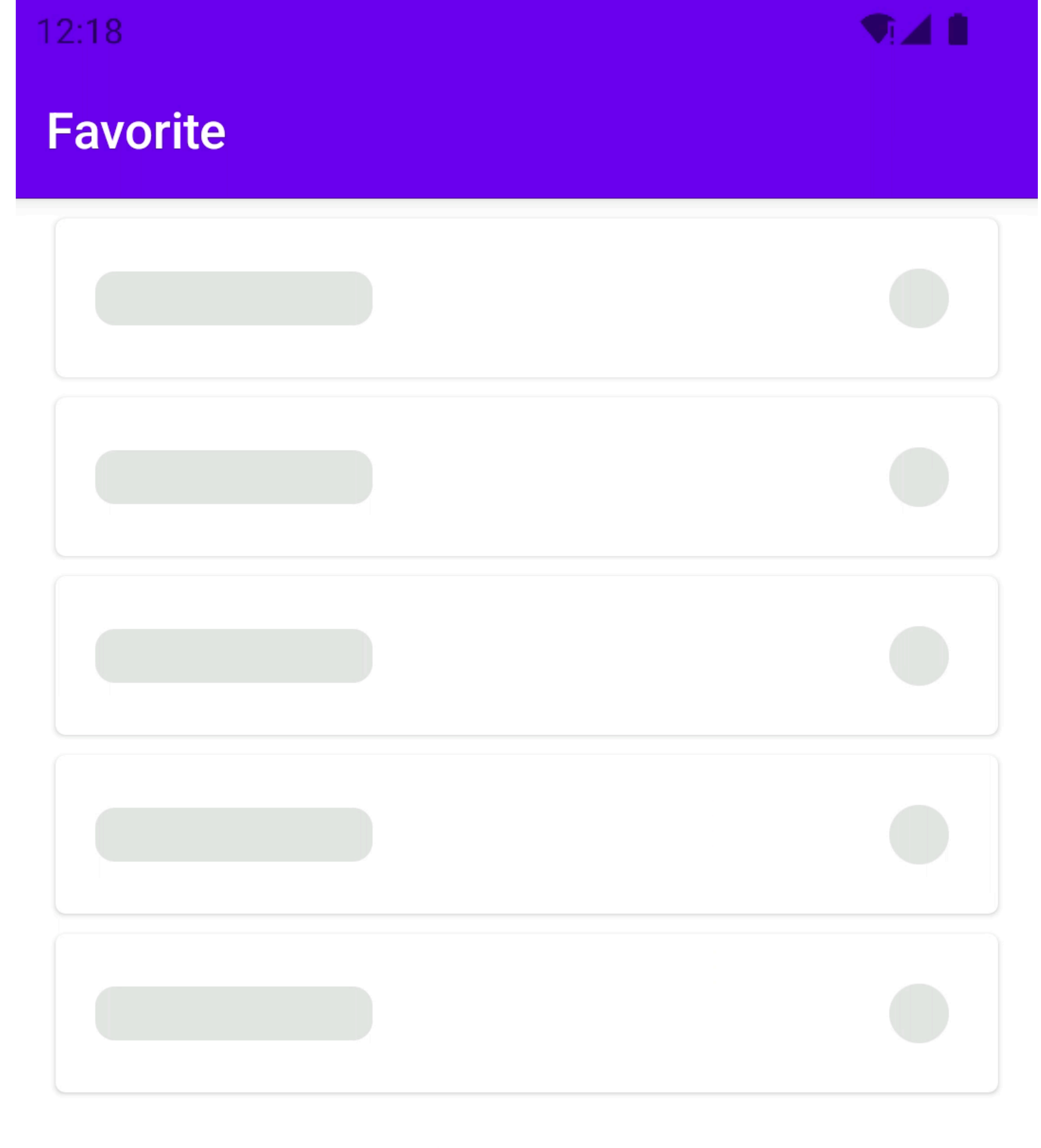
Test process death (PD)

1. Start app
2. Wait success loading
3. Go to home screen
4. Kill app process
5. Return to the app
6. There is loading again!



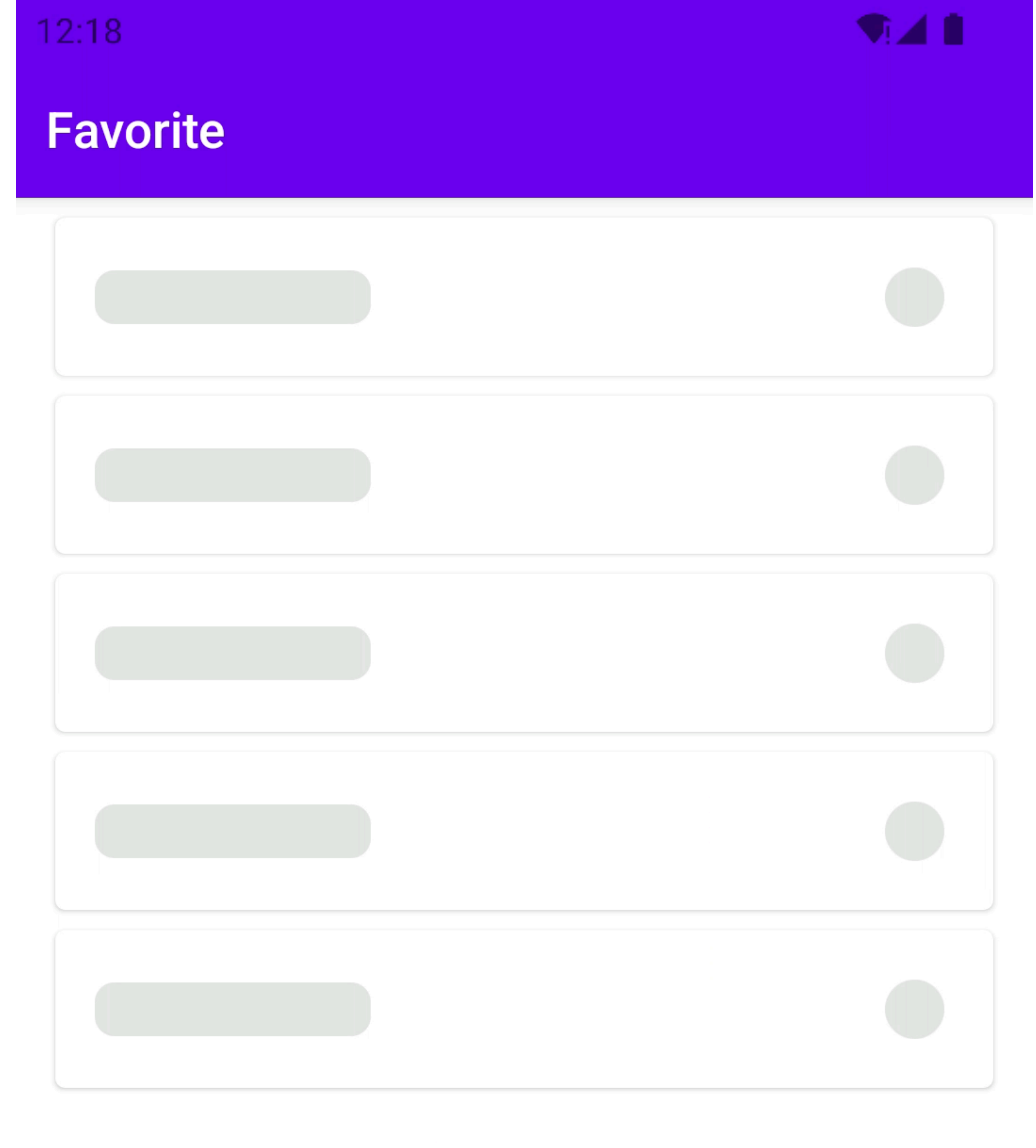
Test process death (PD)

1. Start app
2. Wait success loading
3. Go to home screen
4. Kill app process
5. Return to the app
6. There is loading again!



Test process death (PD)

1. Start app
2. Wait success loading
3. Go to home screen
4. Kill app process
5. Return to the app
6. There is loading again!



Why PD breaks favorite?

```
internal class FavoriteListStore(
    favoriteEffectHandler: FavoriteEffectHandler,
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteListFeature.reducer,
    initialState = State(LCE.Loading()),
    initialEffects = setOf(Eff.Inner.LoadFav, Eff.Inner.ObserveFavUpdates),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

Why PD breaks favorite?

```
internal class FavoriteListStore(
    favoriteEffectHandler: FavoriteEffectHandler,
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteListFeature.reducer,
    initialState = State(LCE.Loading()),
    initialEffects = setOf(Eff.Inner.LoadFav, Eff.Inner.ObserveFavUpdates),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

Fix PD: Initial state in constructor

```
internal class FavoriteListStore(
    effectHandler: FavoriteEffHandler,
    initialState: State = State(LCE.Loading()),
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteListFeature.reducer,
    initialState = initialState,
    initialEffects = setOf(
        Eff.Inner.LoadFav,
        Eff.Inner.ObserveFavUpdates
    ),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

Fix PD: Initial state in constructor

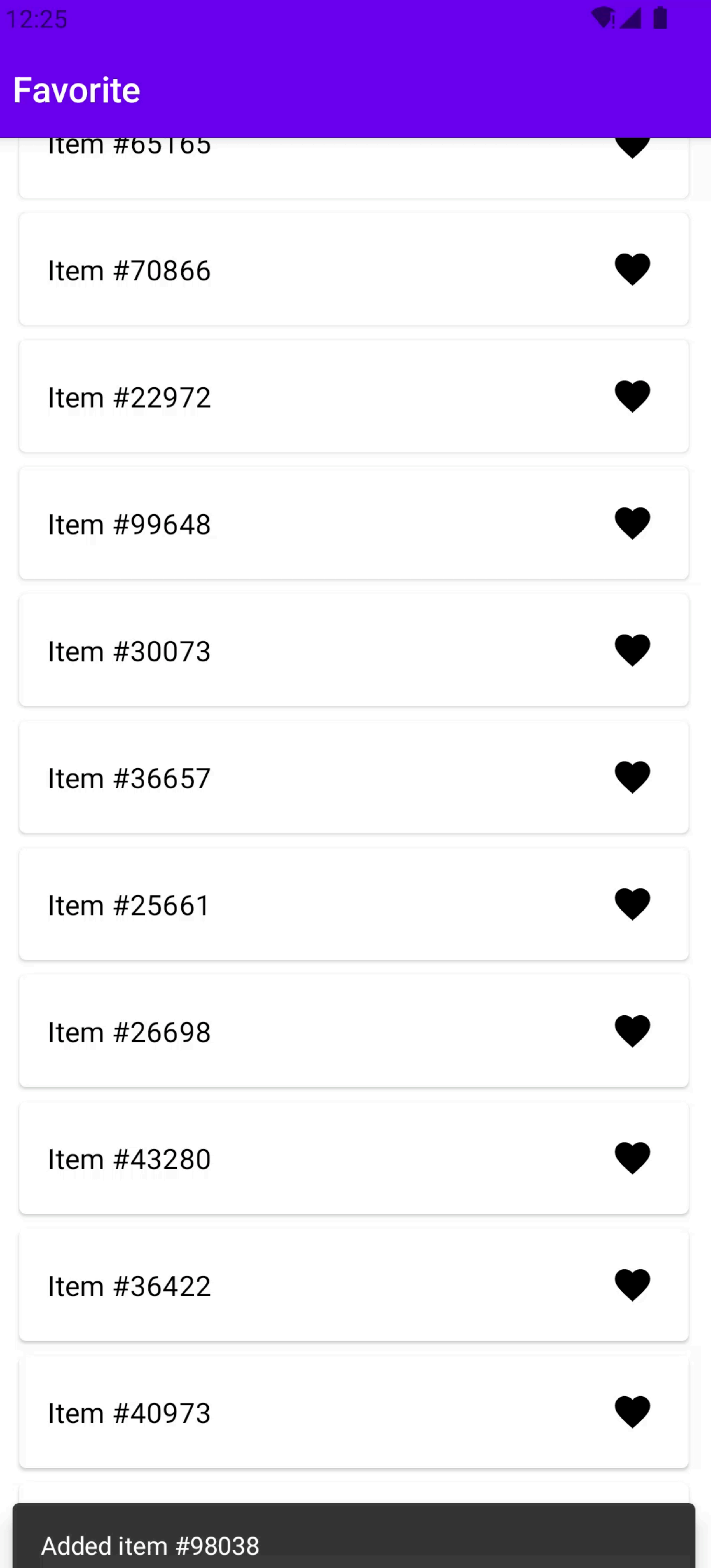
```
internal class FavoriteListStore(
    effectHandler: FavoriteEffHandler,
    initialState: State = State(LCE.Loading()),
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteListFeature.reducer,
    initialState = initialState,
    initialEffects = setOf(
        Eff.Inner.LoadFav,
        Eff.Inner.ObserveFavUpdates
    ),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

Fix PD: Initial state in constructor

```
internal class FavoriteListStore(
    effectHandler: FavoriteEffHandler,
    initialState: State = State(LCE.Loading()),
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteListFeature.reducer,
    initialState = initialState,
    initialEffects = setOf(
        Eff.Inner.LoadFav,
        Eff.Inner.ObserveFavUpdates
    ),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

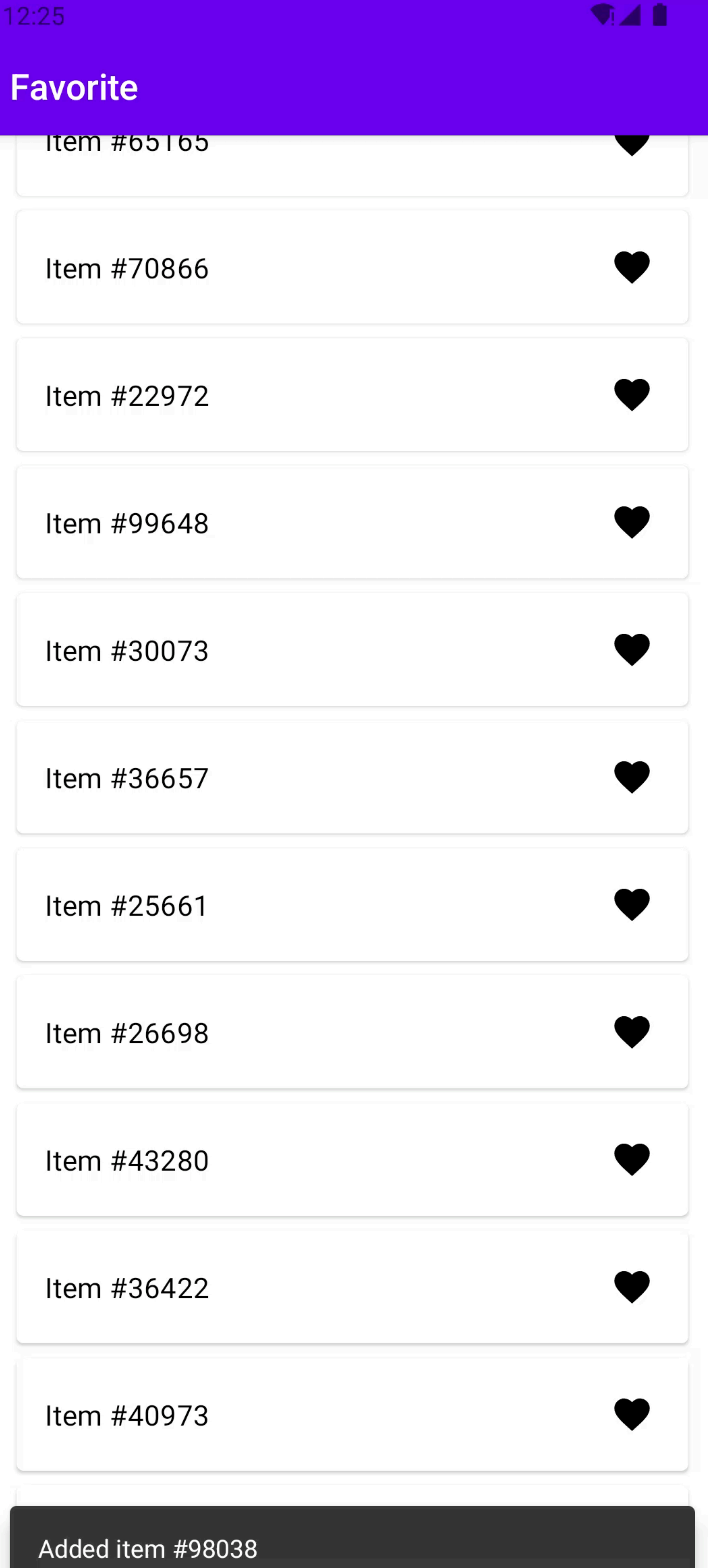
Test PD after Fix

1. App with scrollable loaded list
2. Go to home screen
3. Kill app process
4. Return to the app
5. Correct! But wait
6. There is loading again!



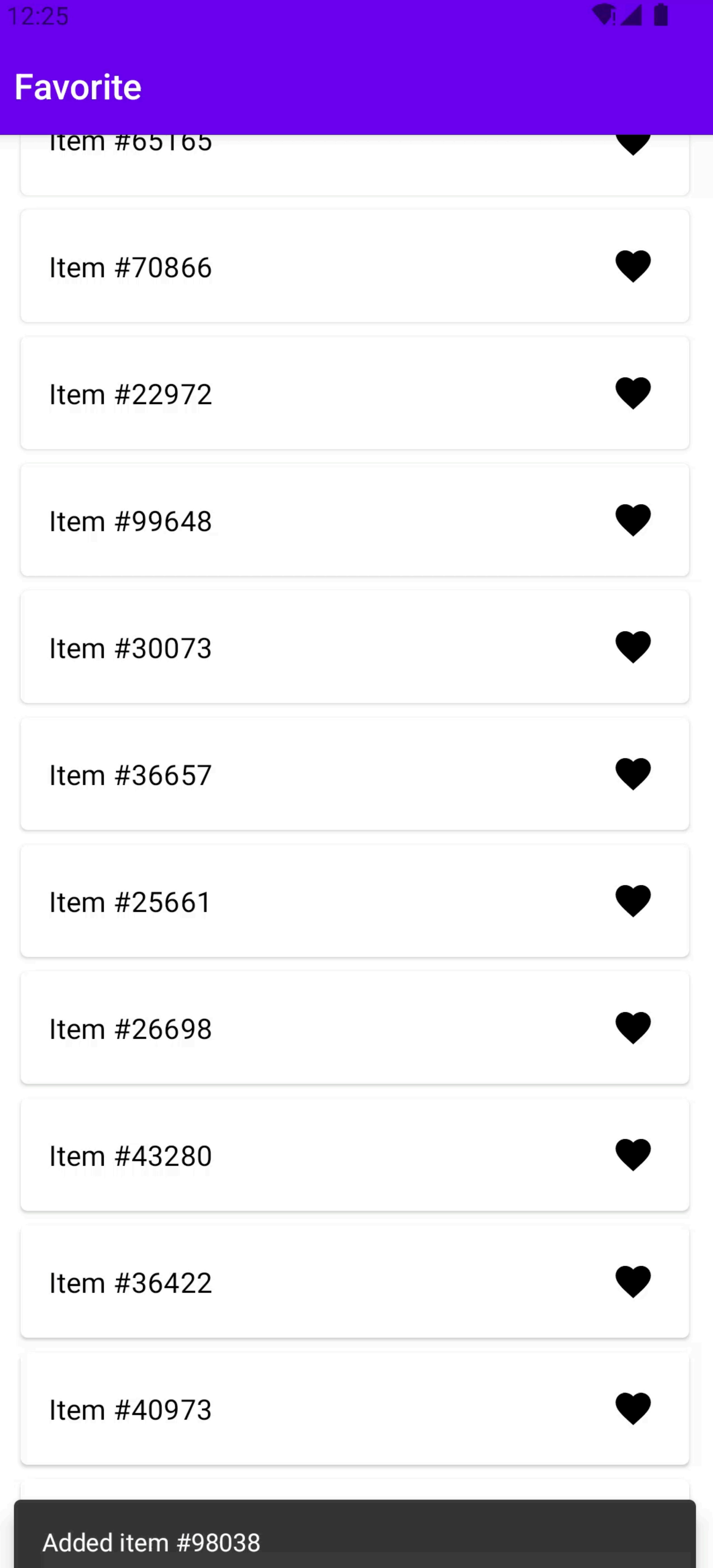
Test PD after Fix

1. **App with scrollable loaded list**
2. Go to home screen
3. Kill app process
4. Return to the app
5. Correct! But wait
6. There is loading again!



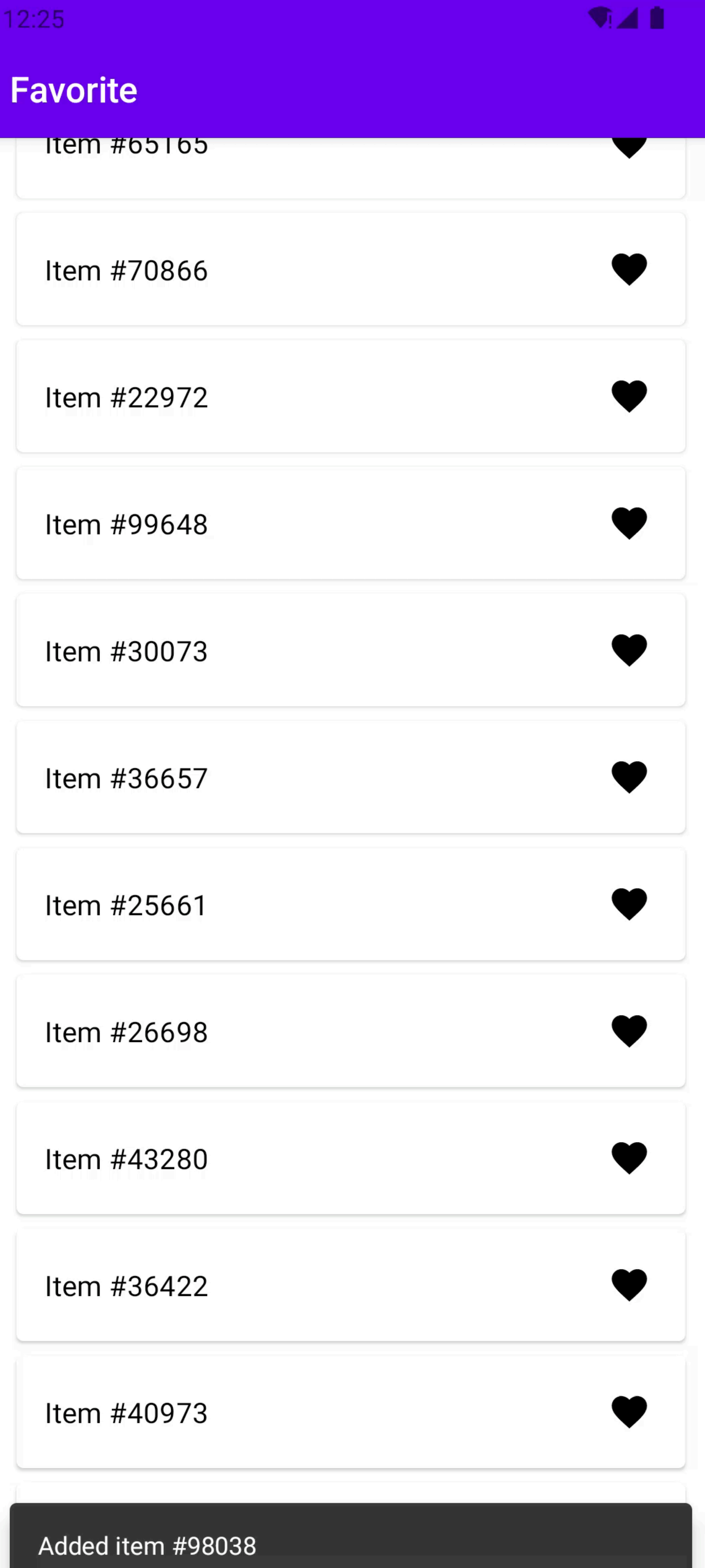
Test PD after Fix

1. **App with scrollable loaded list**
2. **Go to home screen**
3. Kill app process
4. Return to the app
5. Correct! But wait
6. There is loading again!



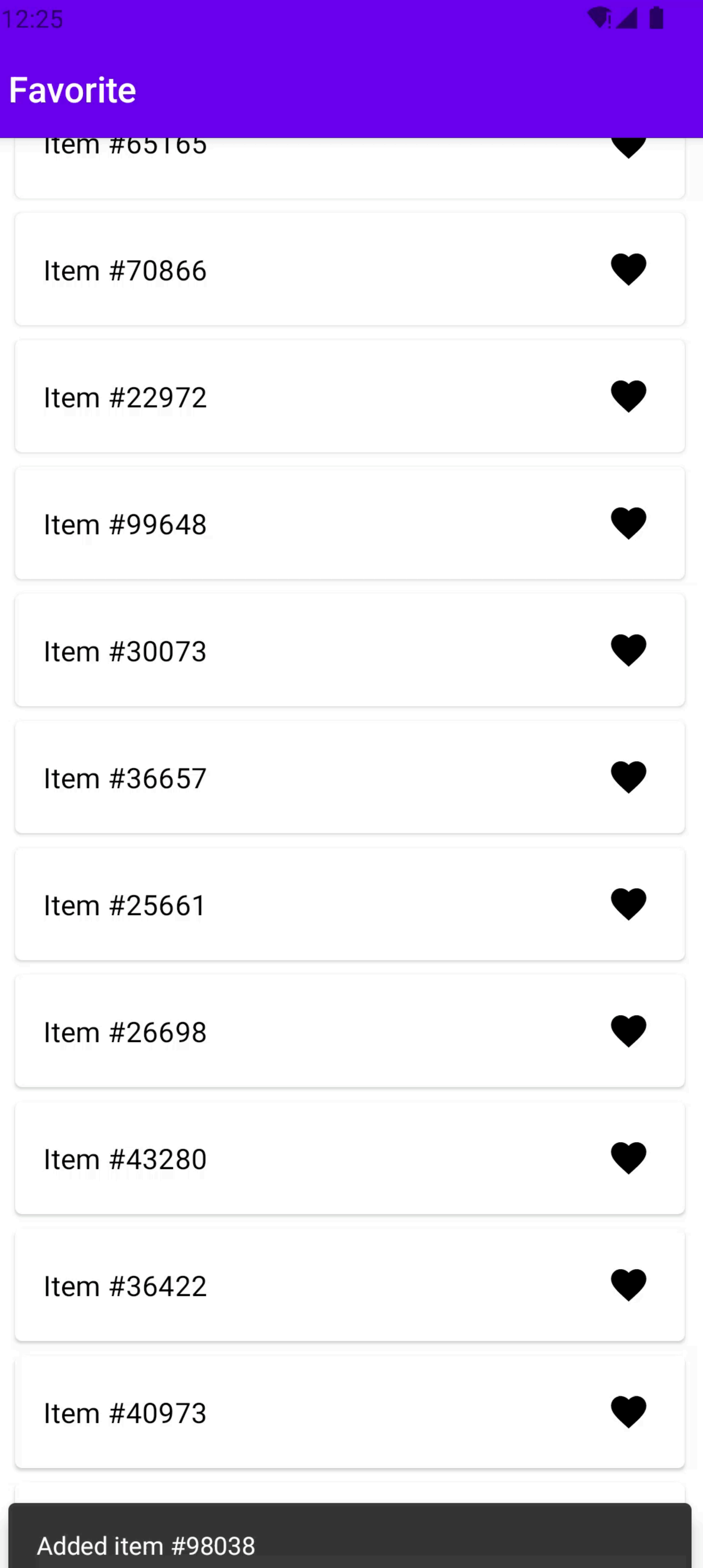
Test PD after Fix

1. App with scrollable loaded list
2. Go to home screen
3. Kill app process
4. Return to the app
5. Correct! But wait
6. There is loading again!



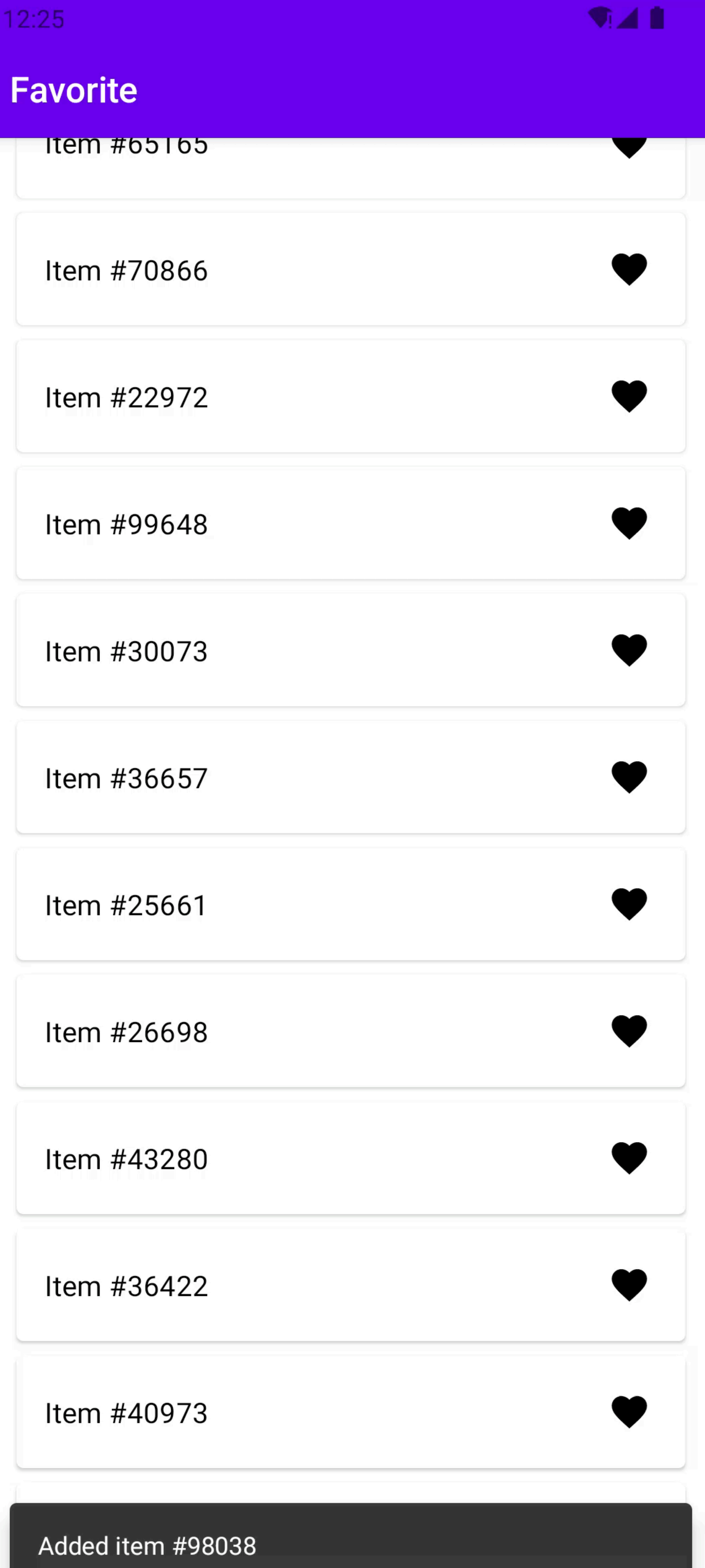
Test PD after Fix

1. App with scrollable loaded list
2. Go to home screen
3. Kill app process
4. Return to the app
5. Correct! But wait
6. There is loading again!



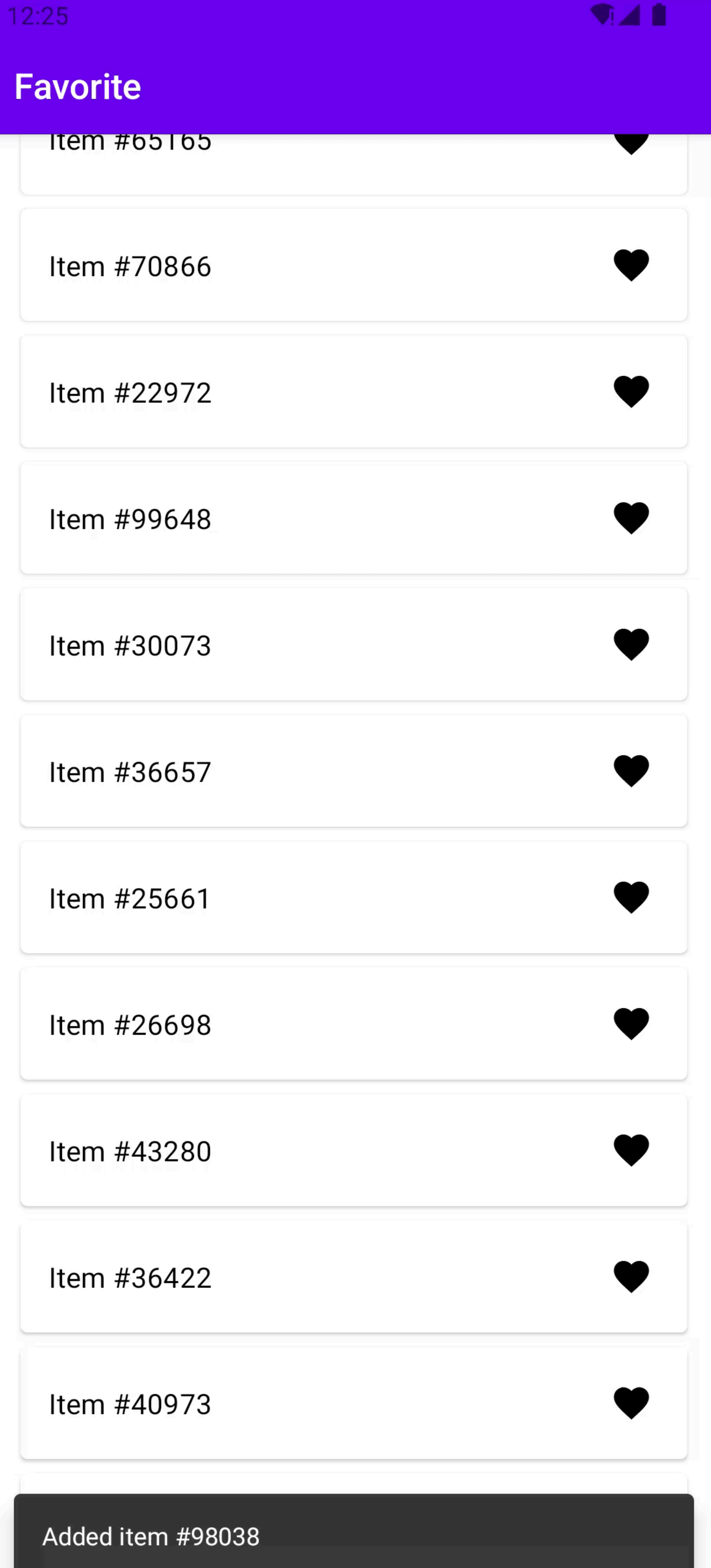
Test PD after Fix

1. App with scrollable loaded list
2. Go to home screen
3. Kill app process
4. Return to the app
5. Correct! But wait
6. There is loading again!



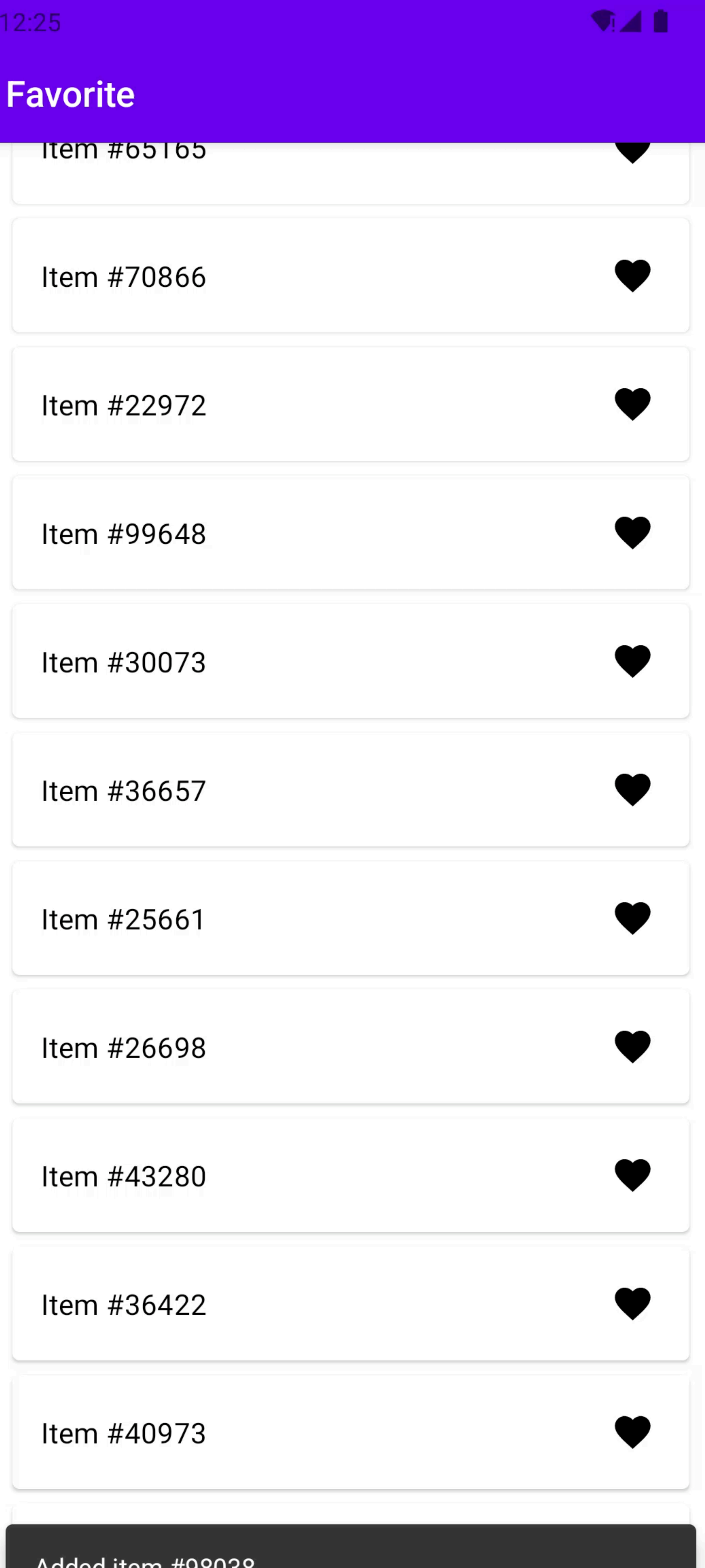
Test PD after Fix

1. App with scrollable loaded list
2. Go to home screen
3. Kill app process
4. Return to the app
5. Correct! But wait
6. There is loading again!



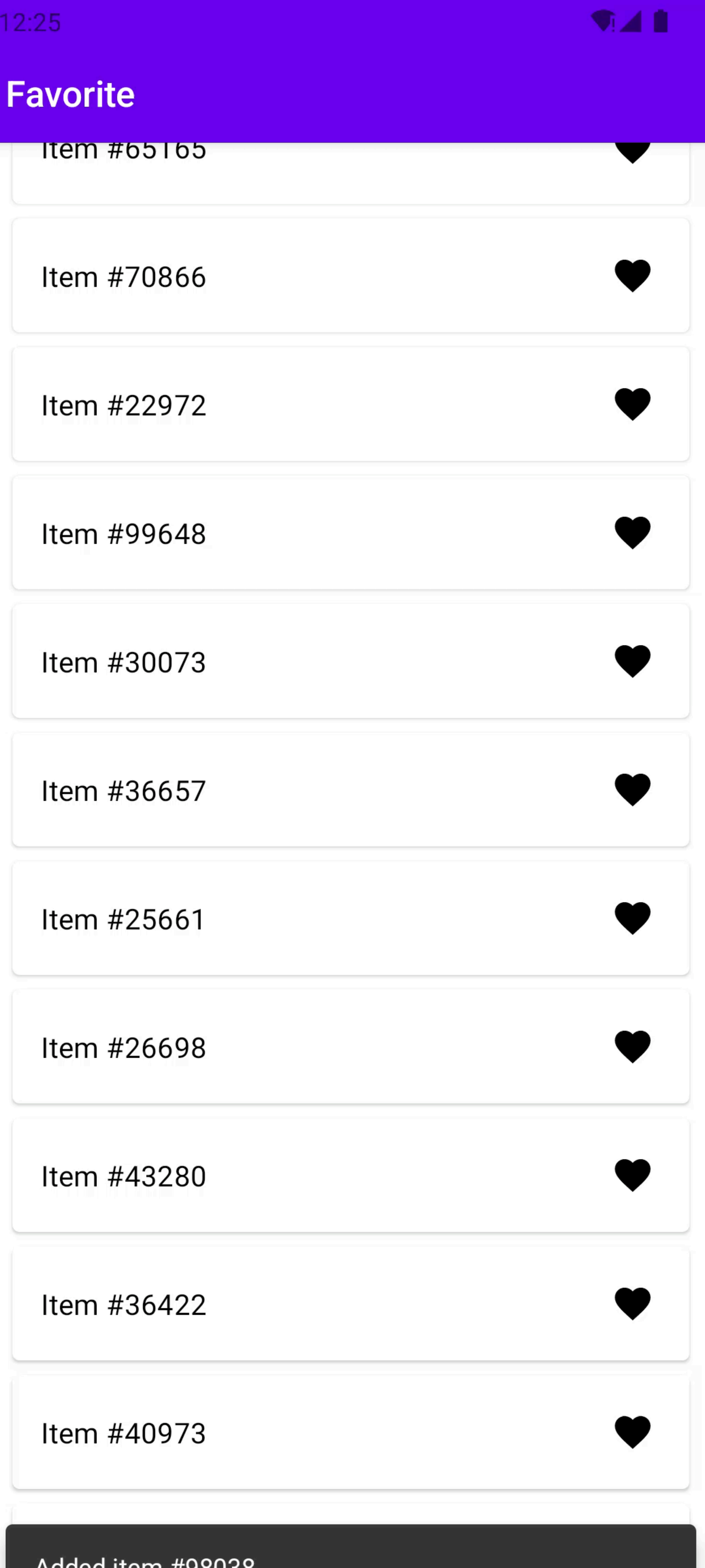
Initial effects

```
internal class FavoriteStore(
    effectHandler: FavoriteEffHandler,
    initialState: State = State(LCE.Loading()),
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteFeature.reducer,
    initialState = initialState,
    initialEffects = setOf(
        Eff.Inner.LoadFav,
        Eff.Inner.ObserveFavUpdates
    ),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```



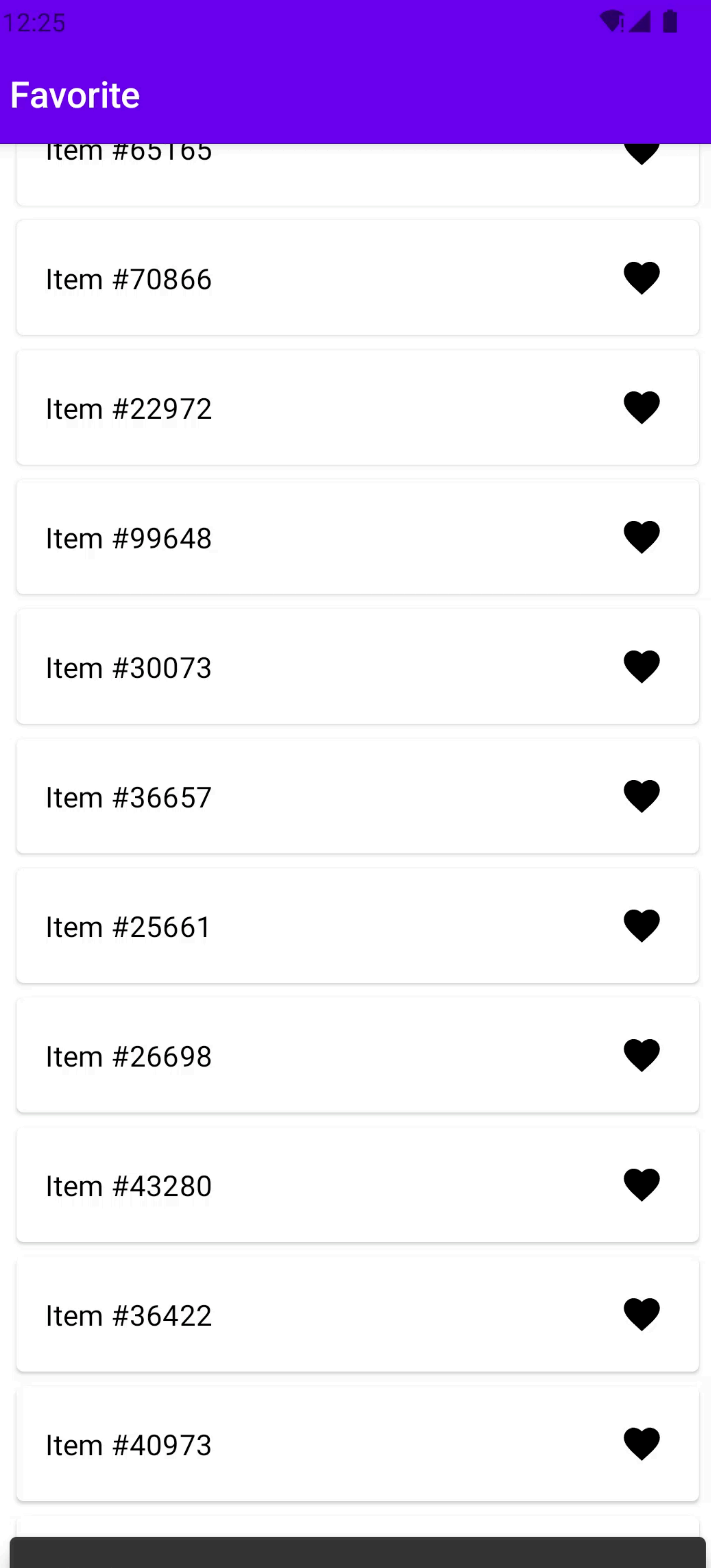
Initial effects

```
internal class FavoriteStore(
    effectHandler: FavoriteEffHandler,
    initialState: State = State(LCE.Loading()),
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteFeature.reducer,
    initialState = initialState,
    initialEffects = setOf(
        Eff.Inner.LoadFav,
        Eff.Inner.ObserveFavUpdates
    ),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```



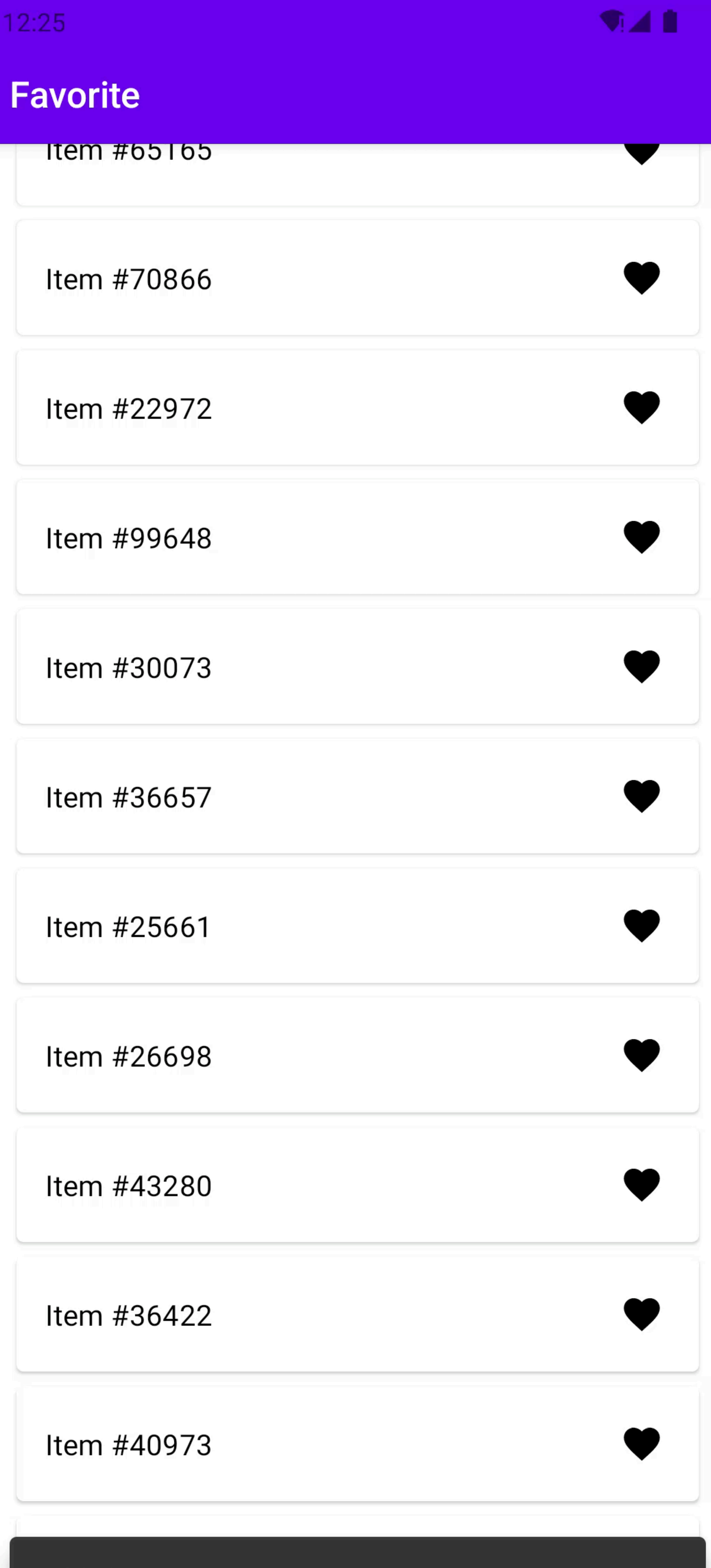
Initial effects

```
internal class FavoriteStore(
    effectHandler: FavoriteEffHandler,
    initialState: State = State(LCE.Loading()),
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteFeature.reducer,
    initialState = initialState,
    initialEffects = setOf(
        Eff.Inner.LoadFav,
        Eff.Inner.ObserveFavUpdates
    ),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```



Initial effects

```
internal class FavoriteStore(
    effectHandler: FavoriteEffHandler,
    initialState: State = State(LCE.Loading()),
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteFeature.reducer,
    initialState = initialState,
    initialEffects = setOf(
        Eff.Inner.LoadFav,
        Eff.Inner.ObserveFavUpdates
    ),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```



Fix PD: Initial effects

```
internal class FavoriteStore(
    effectHandler: FavoriteEffHandler,
    initialState: State = State(LCE.Loading()),
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteFeature.reducer,
    initialState = initialState,
    initialEffects = setOfNotNull(
        Eff.Inner.LoadFav.takeIf {
            initialState == State(LCE.Loading())
        },
        Eff.Inner.ObserveFavUpdates
    ),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

Fix PD: Initial effects

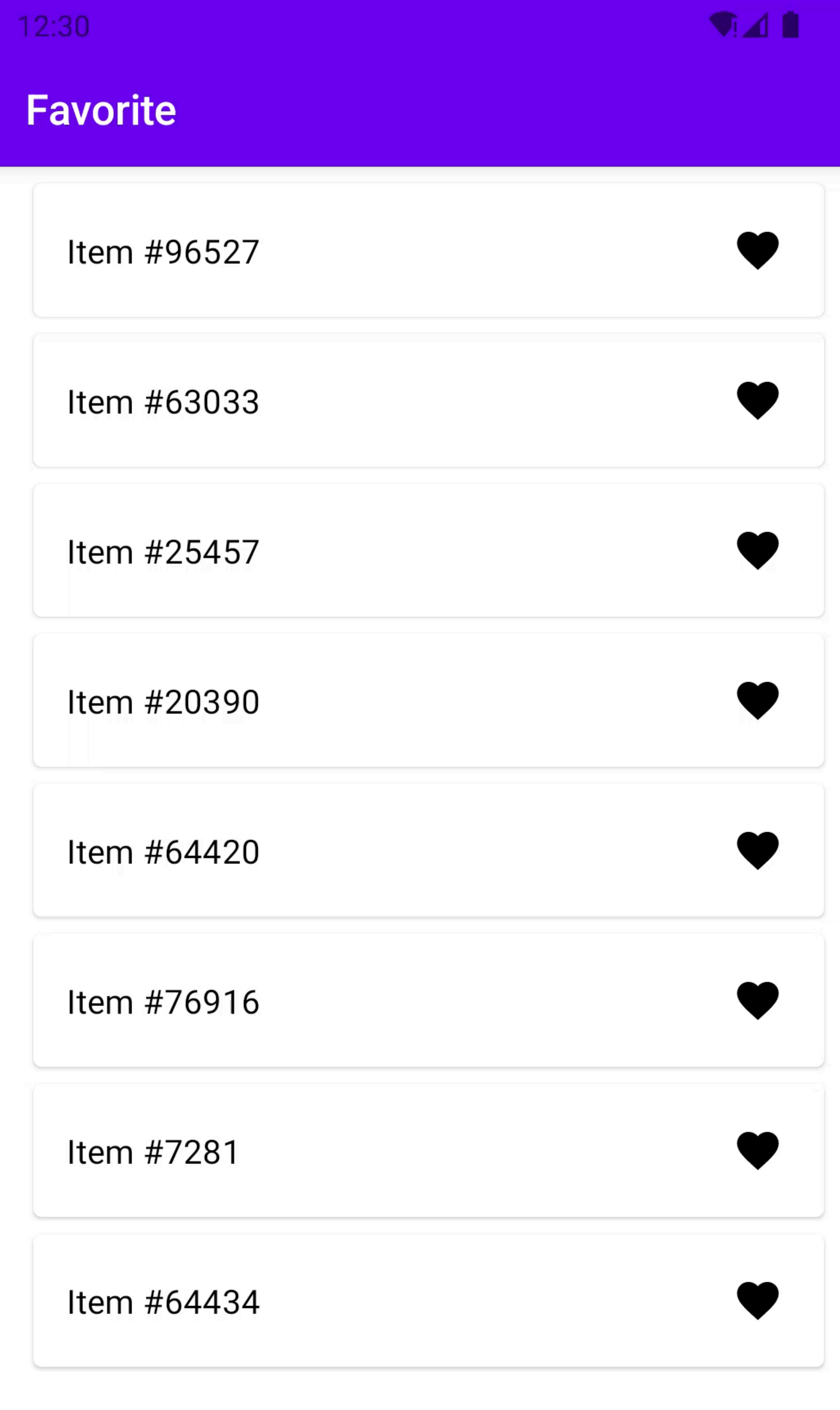
```
internal class FavoriteStore(
    effectHandler: FavoriteEffHandler,
    initialState: State = State(LCE.Loading()),
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteFeature.reducer,
    initialState = initialState,
    initialEffects = setOfNotNull(
        Eff.Inner.LoadFav.takeIf {
            initialState == State(LCE.Loading())
        },
        Eff.Inner.ObserveFavUpdates
    ),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

Fix PD: Initial effects

```
internal class FavoriteStore(
    effectHandler: FavoriteEffHandler,
    initialState: State = State(LCE.Loading()),
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteFeature.reducer,
    initialState = initialState,
    initialEffects = setOfNotNull(
        Eff.Inner.LoadFav.takeIf {
            initialState == State(LCE.Loading())
        },
        Eff.Inner.ObserveFavUpdates
    ),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

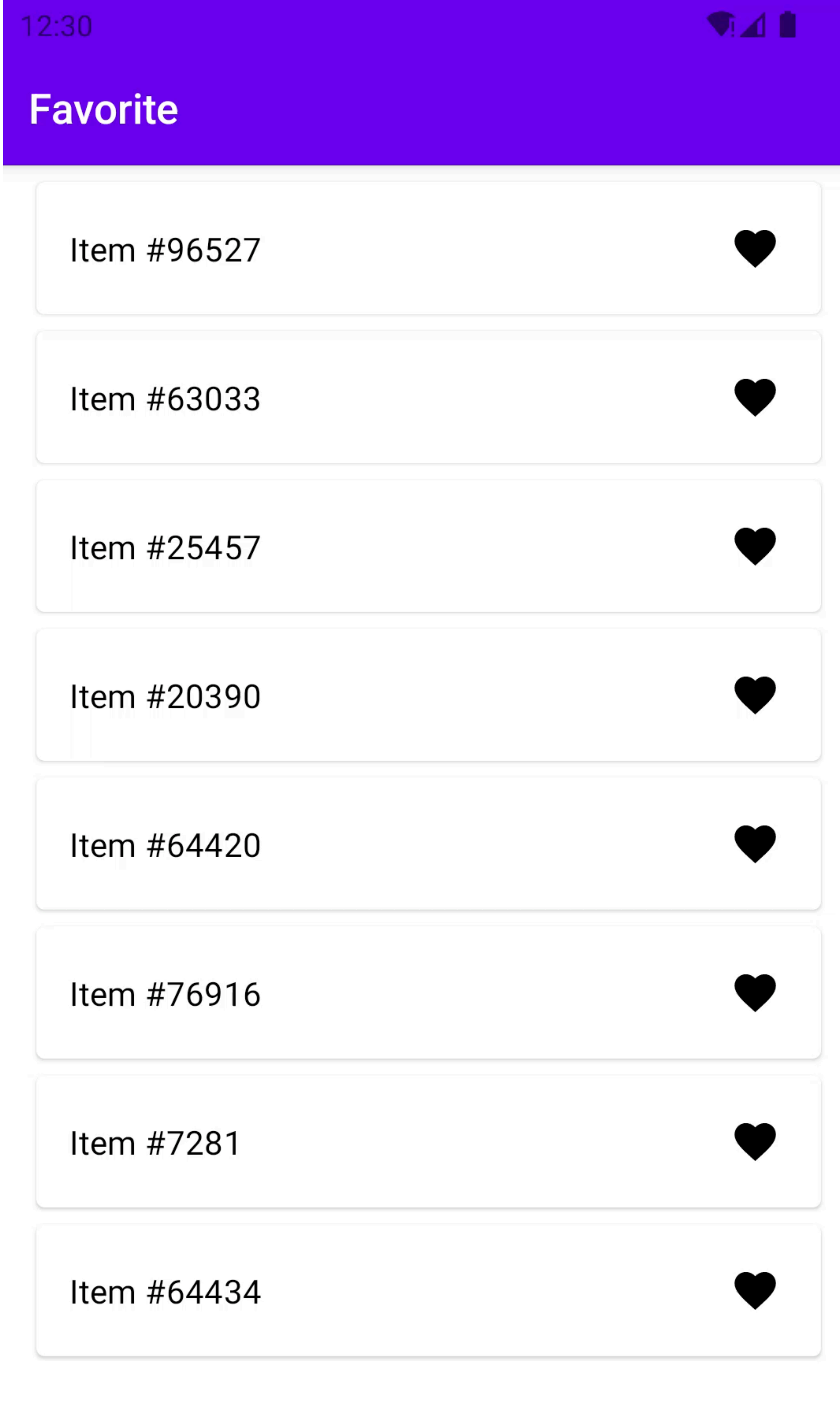
Test PD after 2 Fixes

1. App with scrollable loaded list
2. Go to home screen
3. Kill app process
4. Return to the app
5. Correct! But wait
6. Still correct, congrats!



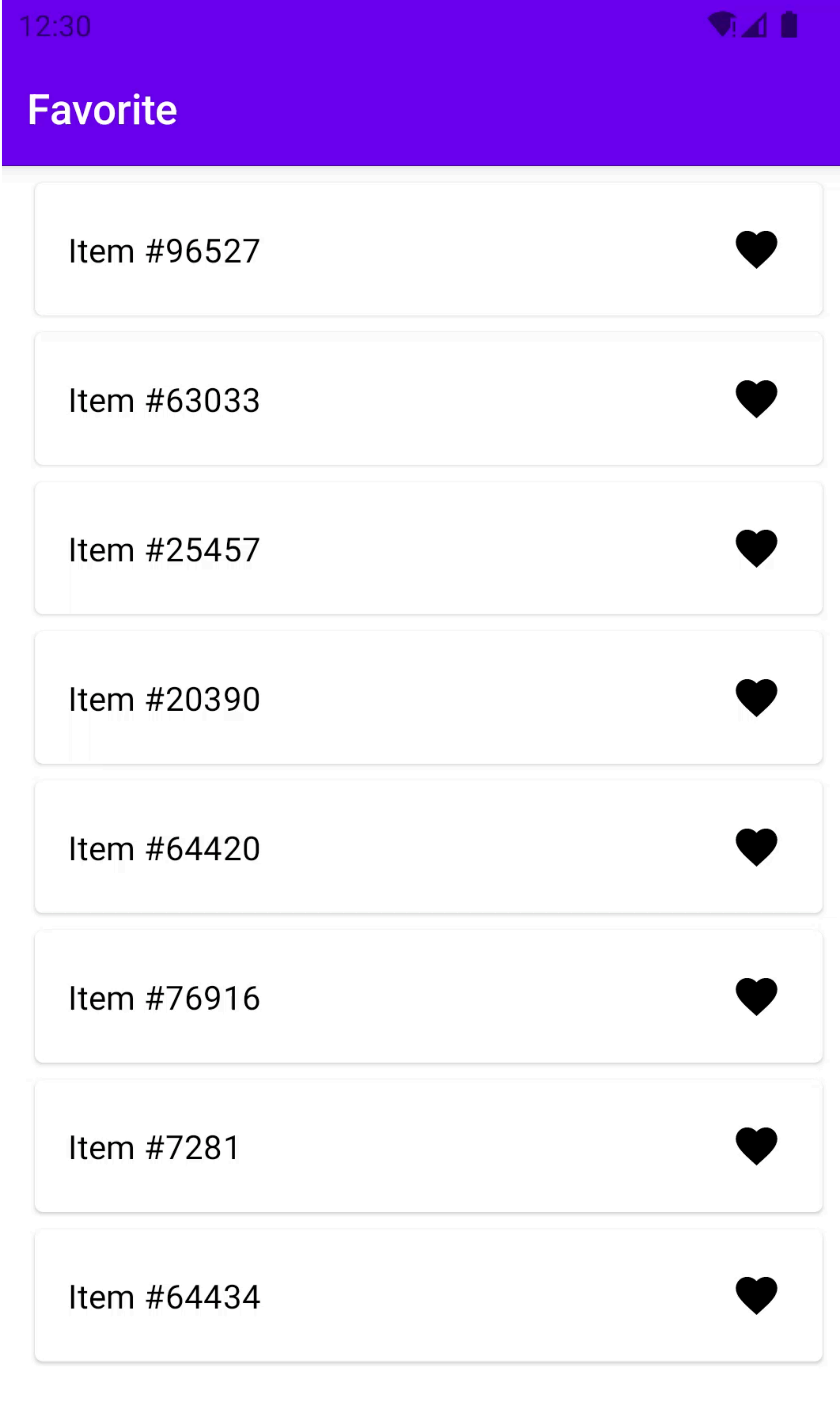
Test PD after 2 Fixes

1. **App with scrollable loaded list**
2. Go to home screen
3. Kill app process
4. Return to the app
5. Correct! But wait
6. Still correct, congrats!



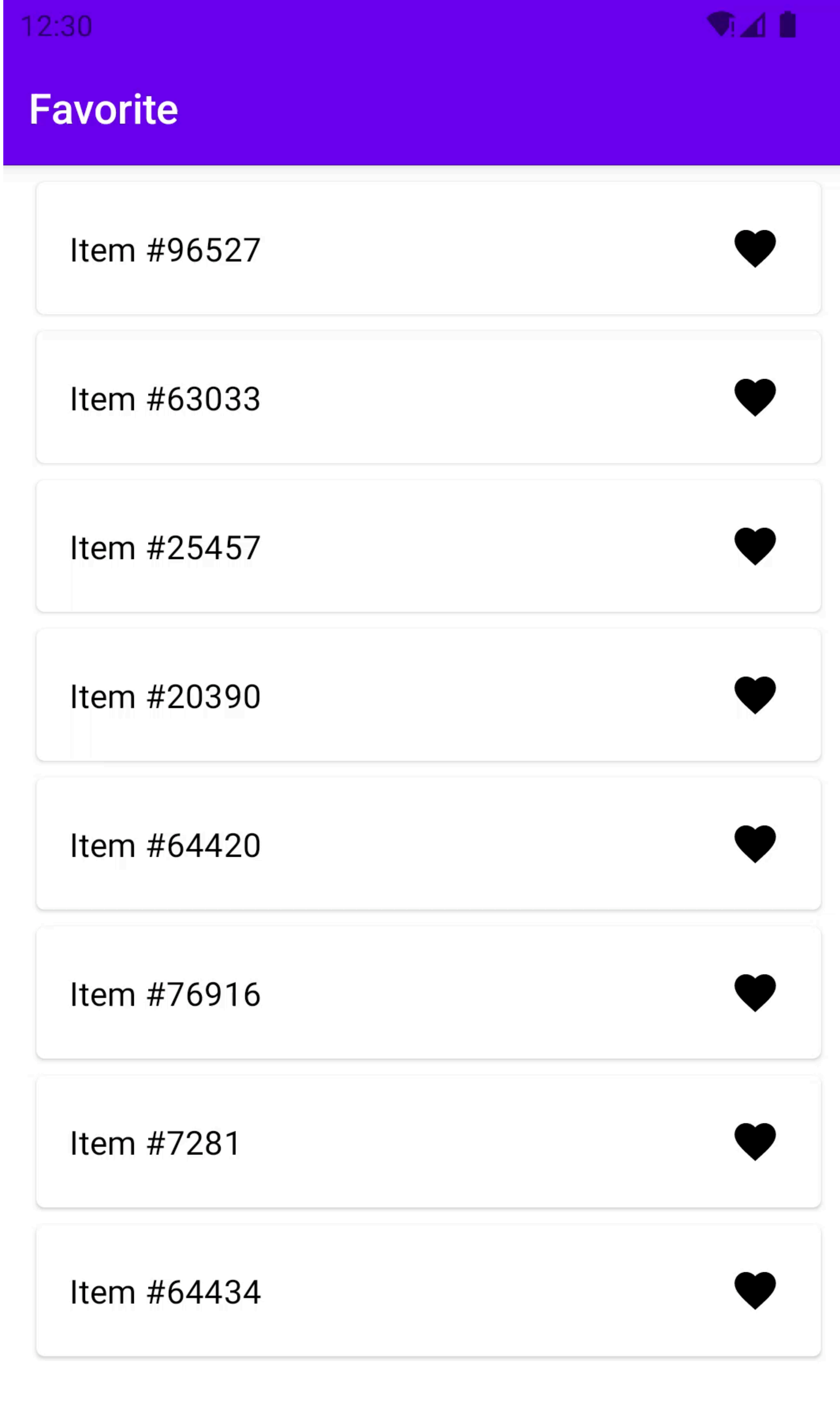
Test PD after 2 Fixes

1. App with scrollable loaded list
2. Go to home screen
3. Kill app process
4. Return to the app
5. Correct! But wait
6. Still correct, congrats!



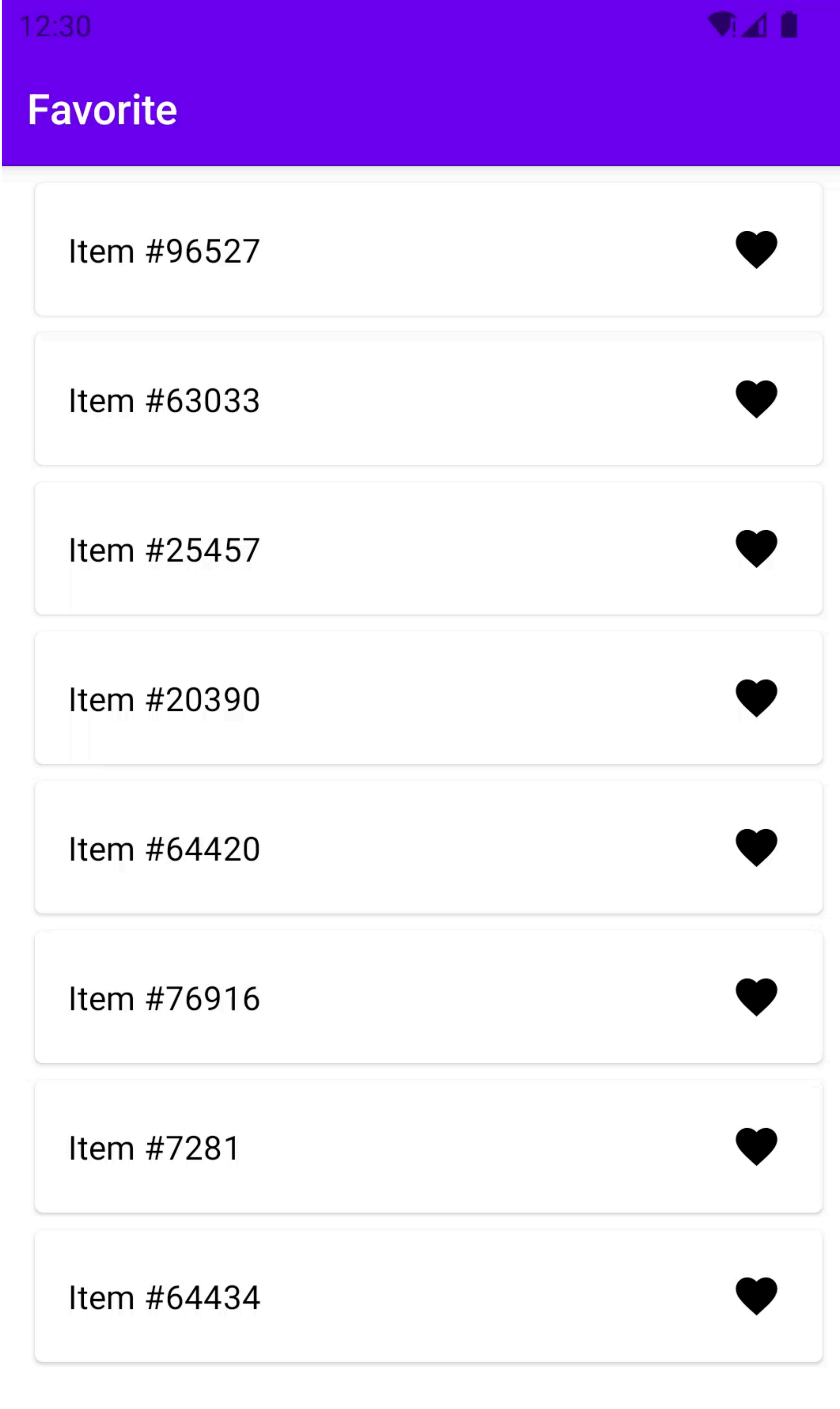
Test PD after 2 Fixes

1. App with scrollable loaded list
2. Go to home screen
3. Kill app process
4. Return to the app
5. Correct! But wait
6. Still correct, congrats!



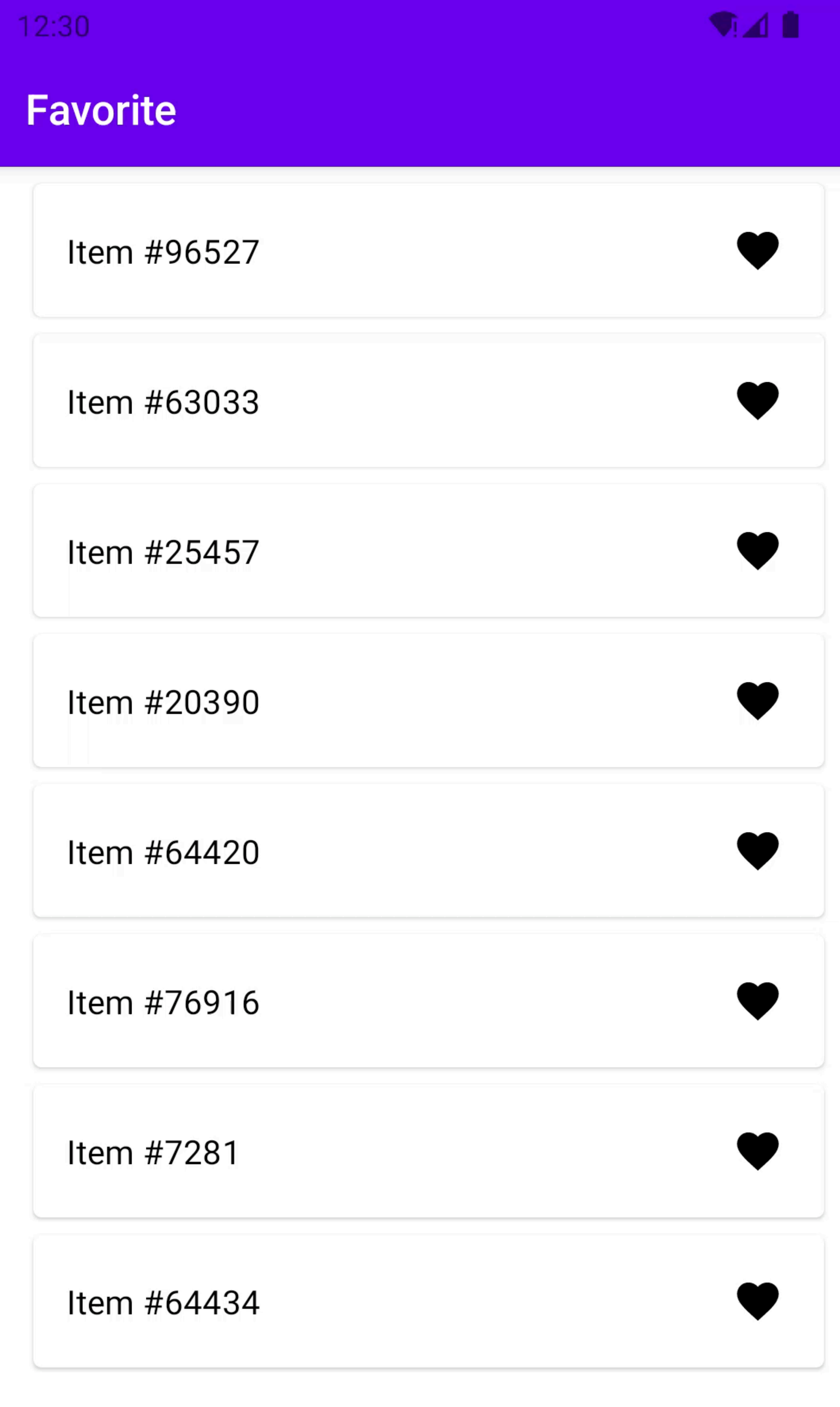
Test PD after 2 Fixes

1. App with scrollable loaded list
2. Go to home screen
3. Kill app process
4. Return to the app
5. Correct! But wait
6. Still correct, congrats!



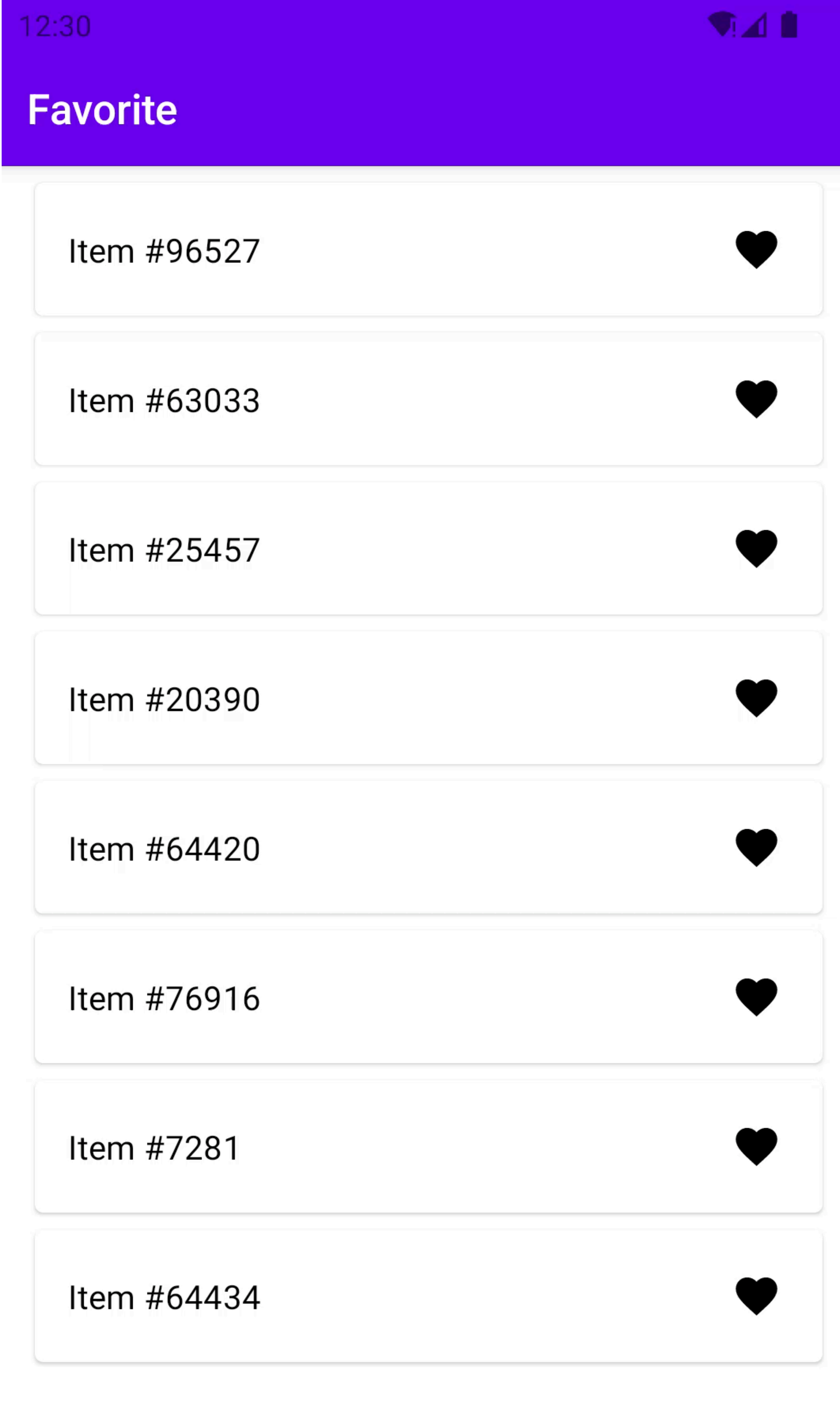
Test PD after 2 Fixes

1. App with scrollable loaded list
2. Go to home screen
3. Kill app process
4. Return to the app
5. Correct! But wait
6. Still correct, congrats!



Test PD after 2 Fixes

1. App with scrollable loaded list
2. Go to home screen
3. Kill app process
4. Return to the app
5. Correct! But wait
6. Still correct, congrats!



PD handling summary

1. Pass saved state as initial
2. Store info about background work in the state
3. Jobs can be restored after PD based on state
4. Eff can be calculated from state

PD handling summary

1. **Pass saved state as initial**
2. Store info about background work in the state
3. Jobs can be restored after PD based on state
4. Eff can be calculated from state

PD handling summary

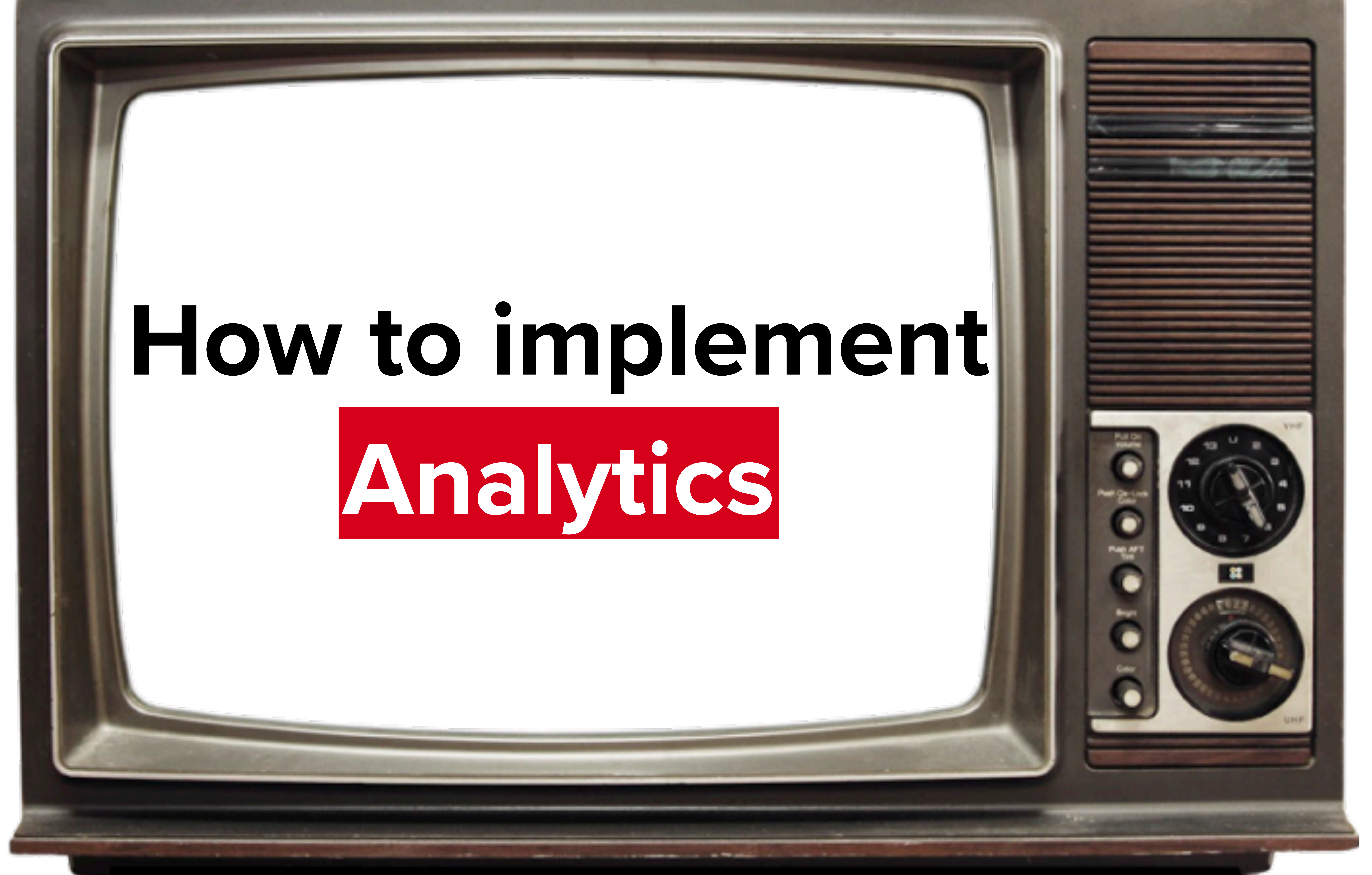
1. Pass saved state as initial
2. Store info about background work in the state
3. Jobs can be restored after PD based on state
4. Eff can be calculated from state

PD handling summary

1. Pass saved state as initial
2. Store info about background work in the state
3. Jobs can be restored after PD based on state
4. Eff can be calculated from state

PD handling summary

1. Pass saved state as initial
2. Store info about background work in the state
3. Jobs can be restored after PD based on state
4. Eff can be calculated from state


How to implement
Analytics

Analytics - Side Eff

Add it as Eff?

```
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        sealed interface Analytics : Inner {
            data class ItemClick(val id: String) : Analytics
            data class ItemFavoriteClick(val id: String) : Analytics
        }
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```

Add it as Eff?

```
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        sealed interface Analytics : Inner {
            data class ItemClick(val id: String) : Analytics
            data class ItemFavoriteClick(val id: String) : Analytics
        }
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```

Add it as Eff?

```
private fun ResultBuilder<State, Eff>.outerReducer(msg: Msg.Outer) {
    when (msg) {
        is Msg.Outer.RemoveFavorite -> {
            ...
            if (!isAlreadyRemoving) {
                eff(Eff.Inner.RemoveItem(msg.id))
            }
            eff(Eff.Inner.Analytics.ItemFavoriteClick(id = msg.id))
        }
        is Msg.Outer.RetryLoad -> {...}
        is Msg.Outer.ItemClick -> {
            eff(
                Eff.Outer.ItemClick(msg.id),
                Eff.Inner.Analytics.ItemClick(msg.id),
            )
        }
    }
}
```

Add it as Eff?

```
private fun ResultBuilder<State, Eff>.outerReducer(msg: Msg.Outer) {
    when (msg) {
        is Msg.Outer.RemoveFavorite -> {
            ...
            if (!isAlreadyRemoving) {
                eff(Eff.Inner.RemoveItem(msg.id))
            }
            eff(Eff.Inner.Analytics.ItemFavoriteClick(id = msg.id))
        }
        is Msg.Outer.RetryLoad -> {...}
        is Msg.Outer.ItemClick -> {
            eff(
                Eff.Outer.ItemClick(msg.id),
                Eff.Inner.Analytics.ItemClick(msg.id),
            )
        }
    }
}
```

Add it as Eff?

```
internal class FavoriteEffHandler(
    private val repository: FavoriteRepository,
) : EffectHandler<Eff.Inner, Msg.Inner> {

    override fun handleEff(eff: Eff.Inner): Flow<Msg.Inner> = when (eff) {
        is Eff.Inner.LoadFav -> flow {...}
        is Eff.Inner.RemoveItem -> flow {...}
        Eff.Inner.ObserveFavUpdates -> ...
        is Eff.Inner.Analytics.ItemClick -> TODO()
        is Eff.Inner.Analytics.ItemFavoriteClick -> TODO()
    }

}
```

Add it as Eff?

```
internal class FavoriteEffHandler(
    private val repository: FavoriteRepository,
) : EffectHandler<Eff.Inner, Msg.Inner> {

    override fun handleEff(eff: Eff.Inner): Flow<Msg.Inner> = when (eff) {
        is Eff.Inner.LoadFav -> flow {...}
        is Eff.Inner.RemoveItem -> flow {...}
        Eff.Inner.ObserveFavUpdates -> ...
        is Eff.Inner.Analytics.ItemClick -> TODO()
        is Eff.Inner.Analytics.ItemFavoriteClick -> TODO()
    }

}
```

I’m done with verbose code

Analytic in Store

```
internal class FavoriteListStore(
    effectHandler: FavoriteEffHandler,
    initialState: State = State(LCE.Loading()),
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteListStore",
    reducer = FavoriteFeature.reducer,
    initialState = initialState,
    initialEffects = setOfNotNull(
        Eff.Inner.LoadFav.takeIf {
            initialState == State(LCE.Loading())
        },
        Eff.Inner.ObserveFavUpdates
    ),
    effectHandlers = listOf(effectHandler.adaptCast())
)
```

Listen for store updates!

```
internal class FavoriteListStore(
    ...
) : CoroutinesStore<Msg, State, Eff>(
    ...
) {

    init {
        coroutinesScope.launch {
            storeUpdates.collect { (msg, oldState, newState, effects) ->
                when (msg) {
                     is Msg.Outer.ItemClick ->
                         favoriteAnalytics.itemClick(msg.id, isFavorite = true)
                    is Msg.Outer.RemoveFavorite ->
                        favoriteAnalytics.changeFavoriteClick(msg.id, desiredFavorite = false)
                    else -> Unit
                }
            }
        }
    }
```

Listen for store updates!

```
internal class FavoriteListStore(
    ...
) : CoroutinesStore<Msg, State, Eff>(
    ...
) {

    init {
        coroutinesScope.launch {
            storeUpdates.collect { (msg, oldState, newState, effects) ->
                when (msg) {
                     is Msg.Outer.ItemClick ->
                         favoriteAnalytics.itemClick(msg.id, isFavorite = true)
                    is Msg.Outer.RemoveFavorite ->
                        favoriteAnalytics.changeFavoriteClick(msg.id, desiredFavorite = false)
                    else -> Unit
                }
            }
        }
    }
```

Listen for store updates!

```
internal class FavoriteListStore(
    ...
) : CoroutinesStore<Msg, State, Eff>(
    ...
) {

    init {
        coroutinesScope.launch {
            storeUpdates.collect { (msg, oldState, newState, effects) ->
                when (msg) {
                     is Msg.Outer.ItemClick ->
                         favoriteAnalytics.itemClick(msg.id, isFavorite = true)
                    is Msg.Outer.RemoveFavorite ->
                        favoriteAnalytics.changeFavoriteClick(msg.id, desiredFavorite = false)
                    else -> Unit
                }
            }
        }
    }
```

What we have learned from Favorite

1. Store for specific task (SRP)
2. Aggregate stores using bindings
3. Lifecycle - handle PD and build store without UI
4. Build reusable components
5. Integrate to the App

What we have learned from Favorite

1. **Store for specific task (SRP)**
2. Aggregate stores using bindings
3. Lifecycle - handle PD and build store without UI
4. Build reusable components
5. Integrate to the App

What we have learned from Favorite

1. Store for specific task (SRP)
2. Aggregate stores using bindings
3. Lifecycle - handle PD and build store without UI
4. Build reusable components
5. Integrate to the App

What we have learned from Favorite

1. Store for specific task (SRP)
2. Aggregate stores using bindings
3. Lifecycle - handle PD and build store without UI
4. Build reusable components
5. Integrate to the App

What we have learned from Favorite

1. Store for specific task (SRP)
2. Aggregate stores using bindings
3. Lifecycle - handle PD and build store without UI
4. Build reusable components
5. Integrate to the App

What we have learned from Favorite

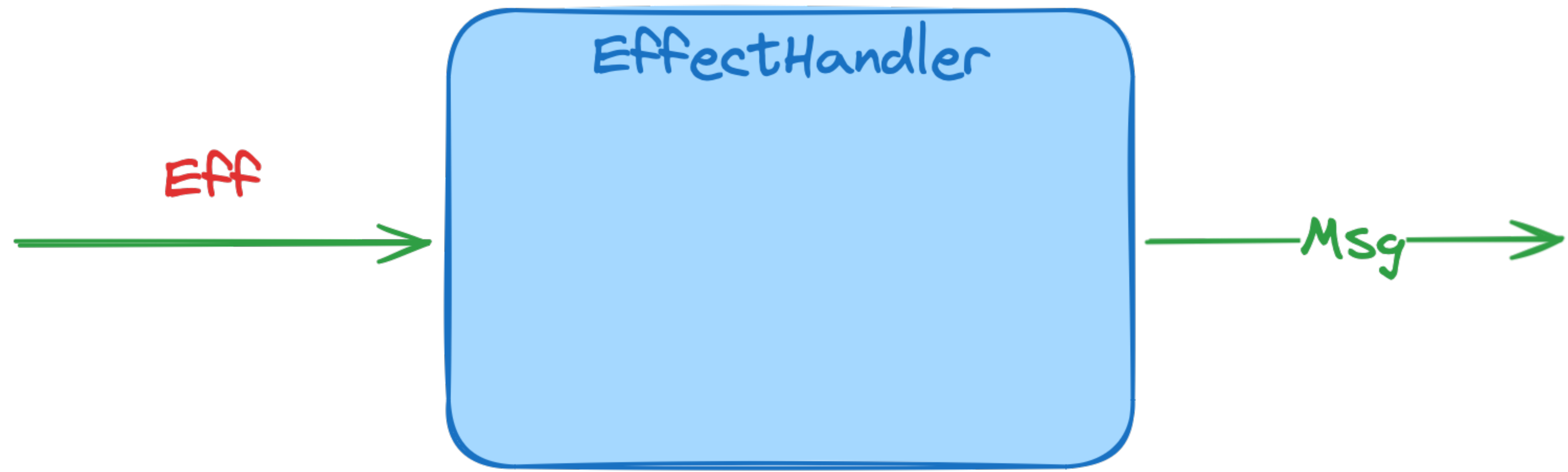
1. Store for specific task (SRP)
2. Aggregate stores using bindings
3. Lifecycle - handle PD and build store without UI
4. Build reusable components
5. Integrate to the App

What left

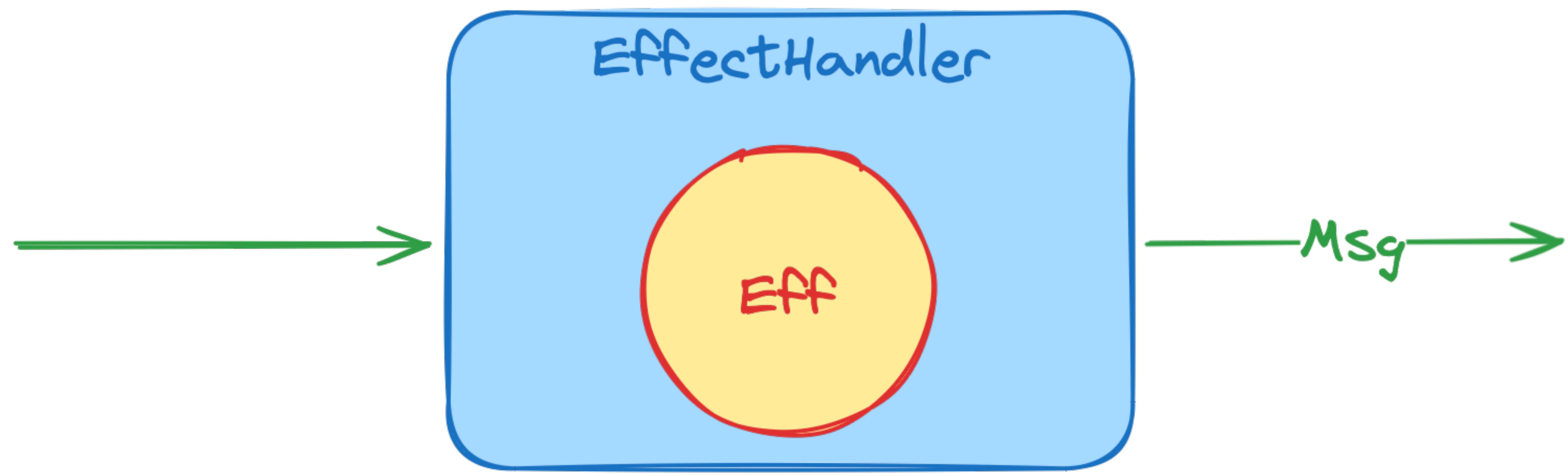
1. Handling long-running tasks
2. UI integration
3. Reusing effect handlers
4. Migration Tips & Tricks

Cancelation of
Long-running task

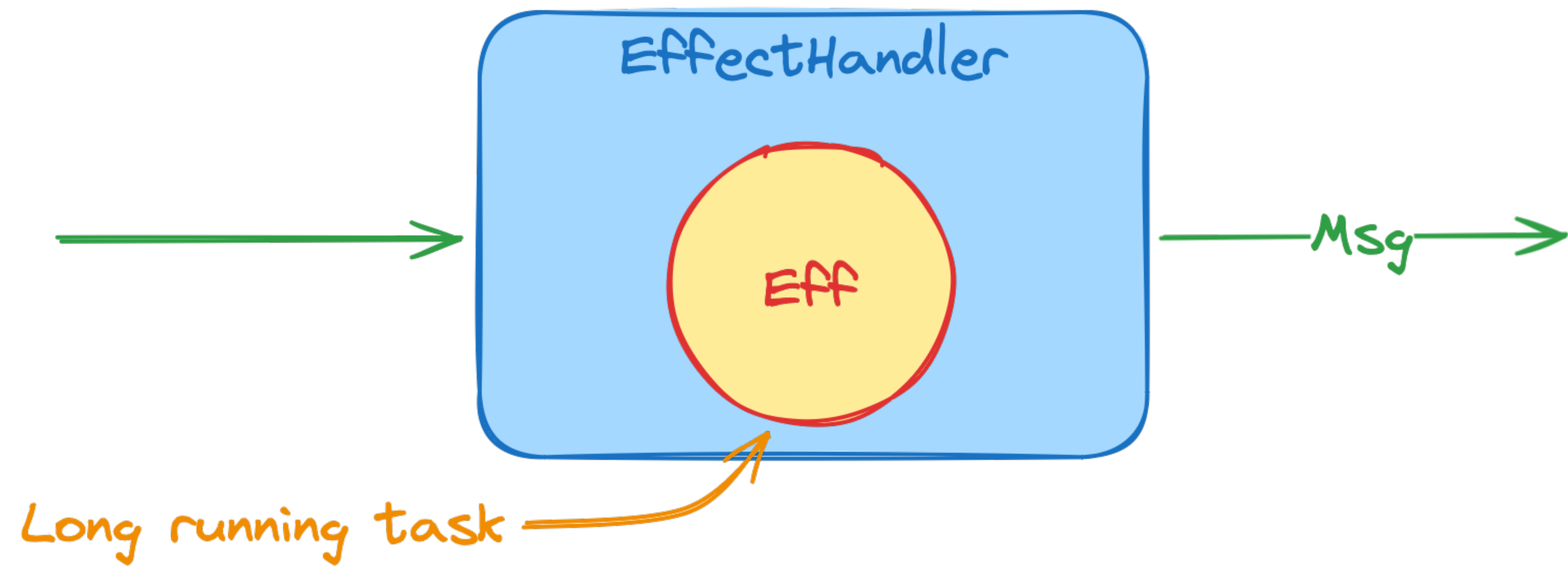
Effects cancelation



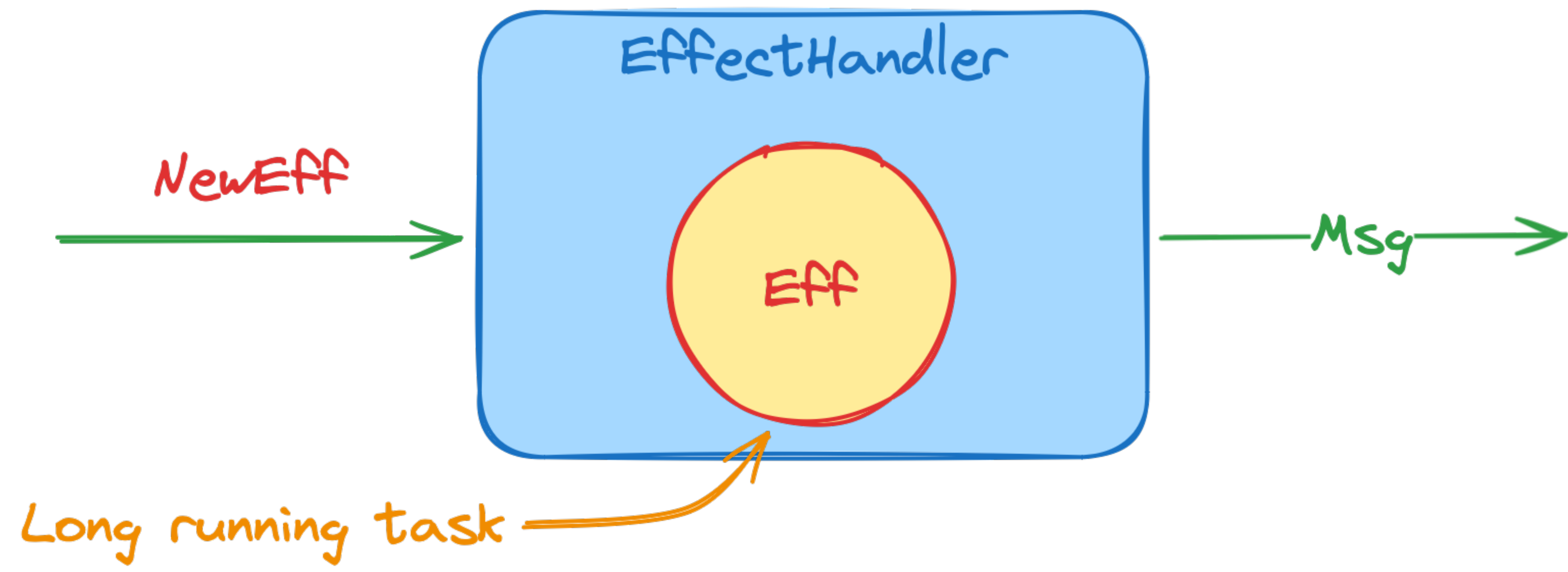
Effects cancelation



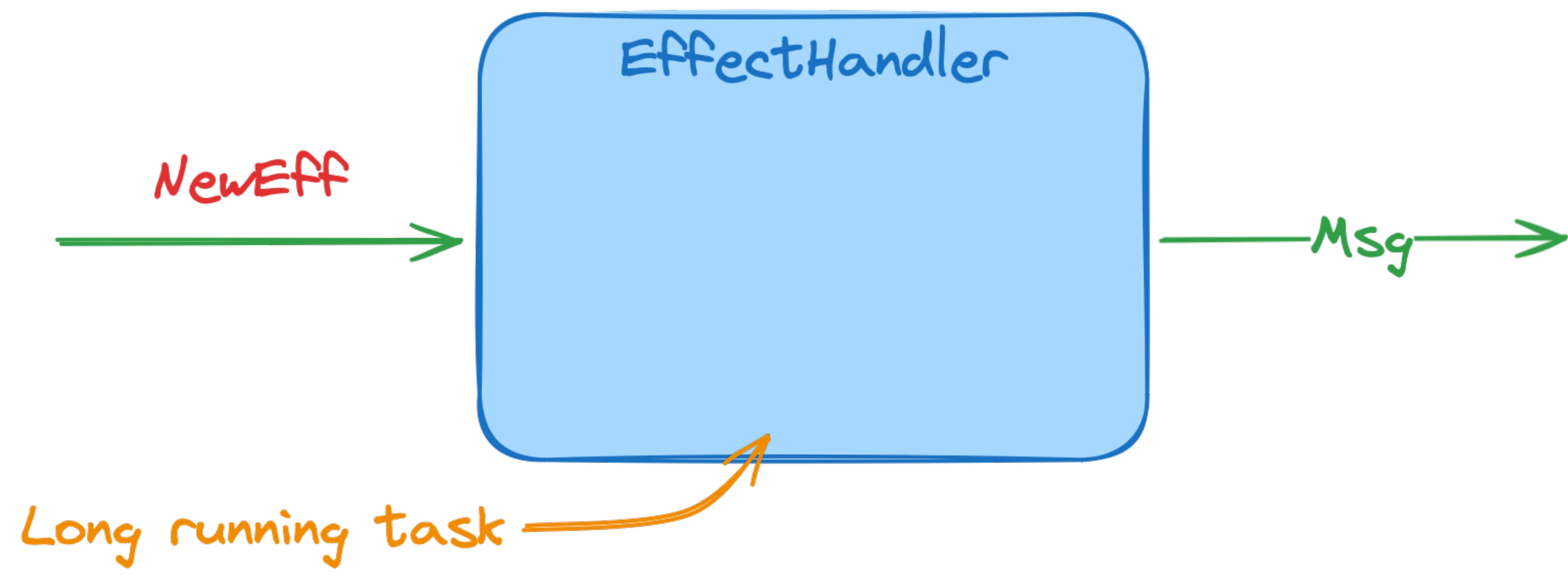
Effects cancelation



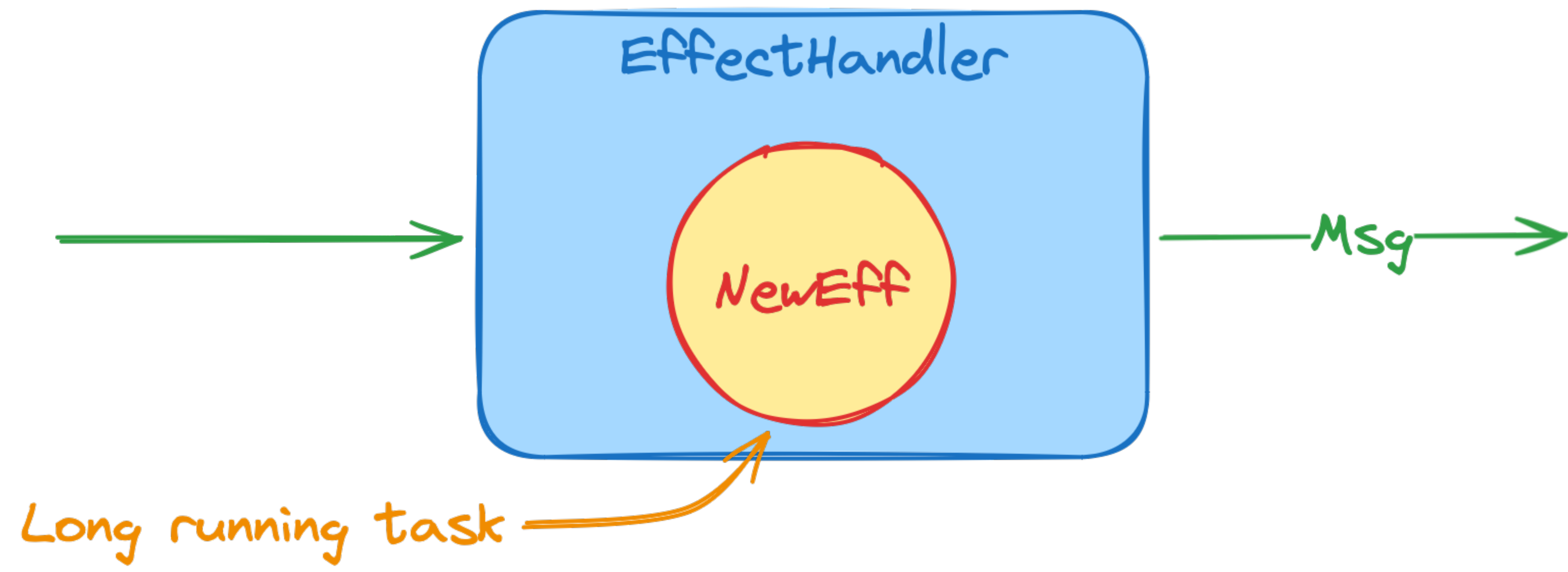
Effects cancelation



Effects cancelation



Effects cancelation



Effects cancelation

```kotlin
@InjectConstructor
class UserCountersIntEffHandler(
    private val userCountersRepository: HrUserCountersRepository,
) : EffectHandler<Eff.Inner, Msg> {

    private val actionManager = ActionManager()

    override fun handleEff(eff: Eff.Inner): Flow<Msg> = when (eff) {
        is Eff.Inner.LoadCounters -> update()
    }

    private fun update(): Flow<Msg> {
        return actionManager.recreateAction {
            flow {
                val counters = withContext(Dispatchers.IO) {
                    userCountersRepository.updateUserCounters().await()
                }
                emit(Msg.SetCounters(counters))
            }
        }
```

Effects cancelation

```
@InjectConstructor
class UserCountersIntEffHandler(
    private val userCountersRepository: HrUserCountersRepository,
) : EffectHandler<Eff.Inner, Msg> {

    private val actionManager = ActionManager()

    override fun handleEff(eff: Eff.Inner): Flow<Msg> = when (eff) {
        is Eff.Inner.LoadCounters -> update()
    }

    private fun update(): Flow<Msg> {
        return actionManager.recreateAction {
            flow {
                val counters = withContext(Dispatchers.IO) {
                    userCountersRepository.updateUserCounters().await()
                }
                emit(Msg.SetCounters(counters))
            }
        }
```

Effects cancelation

```
@InjectConstructor
class UserCountersIntEffHandler(
    private val userCountersRepository: HrUserCountersRepository,
) : EffectHandler<Eff.Inner, Msg> {

    private val actionManager = ActionManager()

    override fun handleEff(eff: Eff.Inner): Flow<Msg> = when (eff) {
        is Eff.Inner.LoadCounters -> update()
    }

    private fun update(): Flow<Msg> {
        return actionManager.recreateAction {
            flow {
                val counters = withContext(Dispatchers.IO) {
                    userCountersRepository.updateUserCounters().await()
                }
                emit(Msg.SetCounters(counters))
            }
        }
```

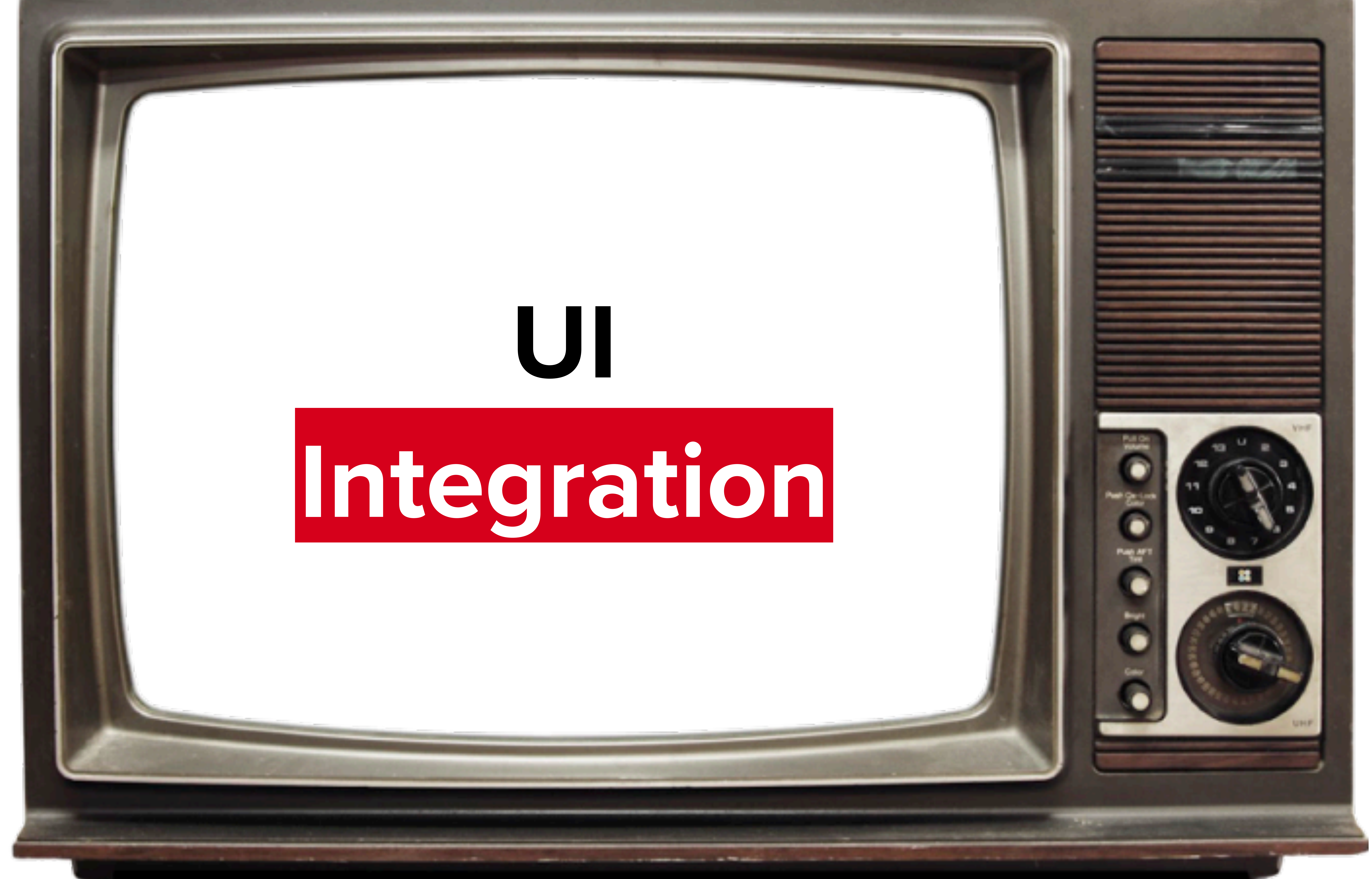
Effects cancelation

```kotlin
@InjectConstructor
class UserCountersIntEffHandler(
    private val userCountersRepository: HrUserCountersRepository,
) : EffectHandler<Eff.Inner, Msg> {

    private val actionManager = ActionManager()

    override fun handleEff(eff: Eff.Inner): Flow<Msg> = when (eff) {
        is Eff.Inner.LoadCounters -> update()
    }

    private fun update(): Flow<Msg> {
        return actionManager.recreateAction {
            flow {
                val counters = withContext(Dispatchers.IO) {
                    userCountersRepository.updateUserCounters().await()
                }
                emit(Msg.SetCounters(counters))
            }
        }
```

Effects cancelation

```
/**
 * Allows to execute requests for [EffectHandler] implementations in a switching
manner. Each request
 * will cancel the previous one.
 *
 * Example:
 * ```
 * private val actionManager = ActionManager() io.github.ikarenkov.kombucha.eff_handler
 *
 *
 * override fun handleEff(eff: Eff.Int): Flow<Msg> = when (eff) {
 *      is Eff.MyEff -> actionManager.recreateAction {
 *          flow { ... }
 *      }
 * }
 * ```
 */
class ActionManager {
```

kombucha/core/src/commonMain/kotlin/io/github/ikarenkov/kombucha/eff_handler/ActionManager.kt


UI
Integration

Msg

```
sealed interface Msg {

    sealed interface Outer : Msg {
        data class ItemClick(val id: String) : Outer
        data class RemoveFavorite(val id: String) : Outer
        data object RetryLoad : Outer
    }

    sealed interface Inner : Msg {

        data class AddItem(val item: FavoriteItem) : Inner
        data class ItemLoadingResult(
            val result: Result<List<FavoriteItem>>
        ) : Inner
        sealed interface ItemRemoveResult : Inner {...}
    }
}
```

10:47

Favorite

Msg

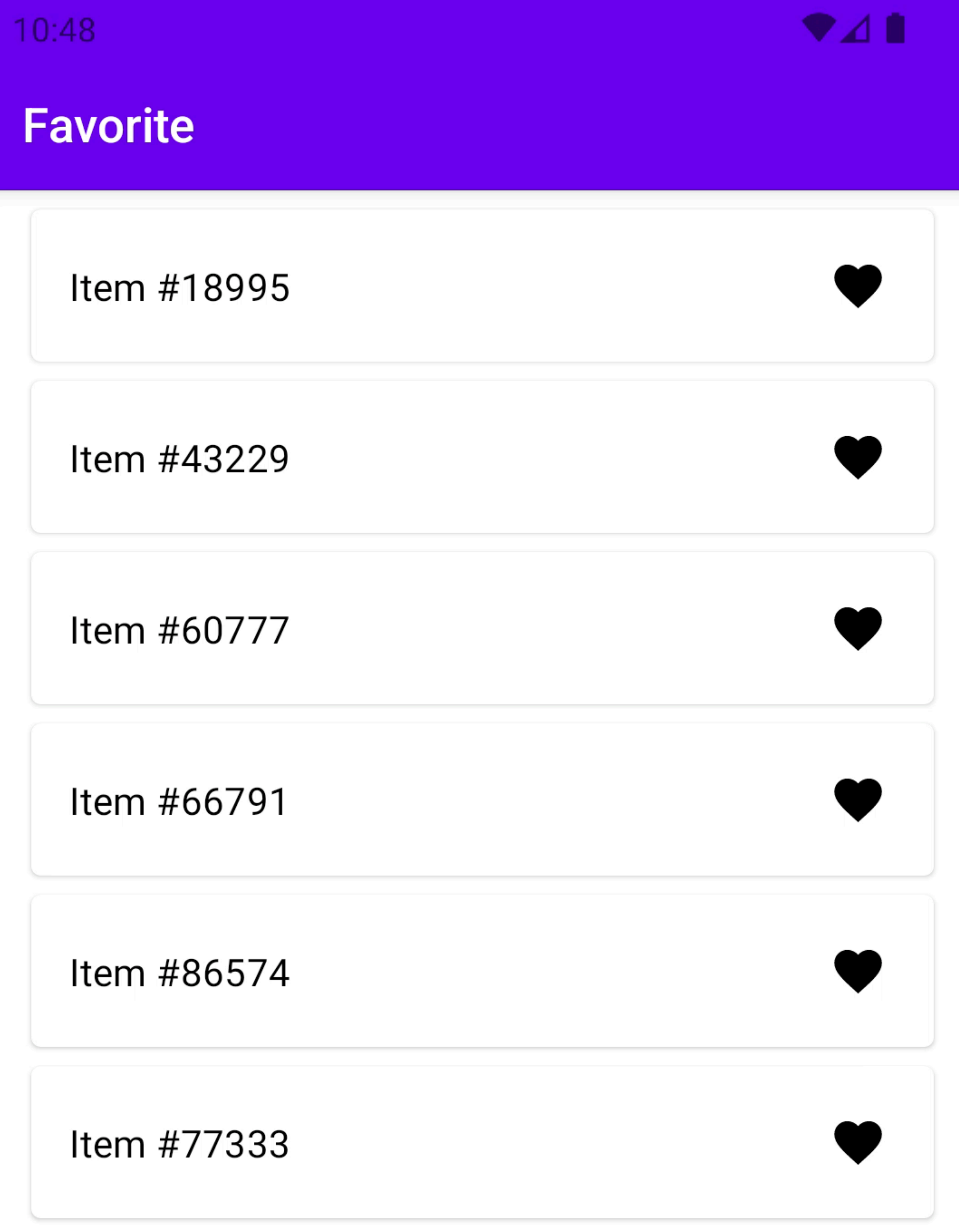
```kotlin
sealed interface Msg {

    sealed interface Outer : Msg {
        data class ItemClick(val id: String) : Outer
        data class RemoveFavorite(val id: String) : Outer
        data object RetryLoad : Outer
    }

    sealed interface Inner : Msg {

        data class AddItem(val item: FavoriteItem) : Inner
        data class ItemLoadingResult(
            val result: Result<List<FavoriteItem>>
        ) : Inner
        sealed interface ItemRemoveResult : Inner {...}
    }
}
```

10:47

Favorite

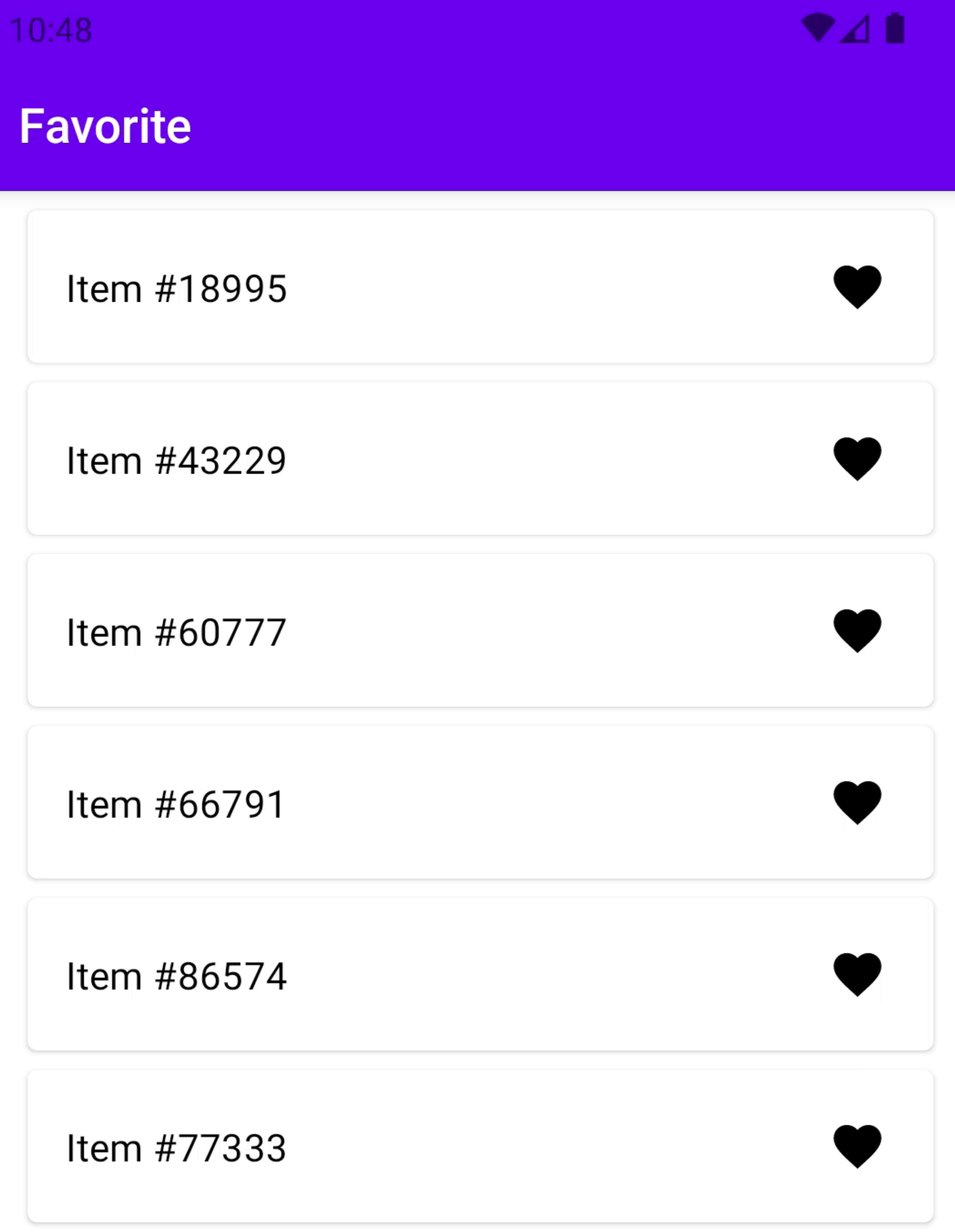
Msg

```
sealed interface Msg {

    sealed interface Outer : Msg {
        data class ItemClick(val id: String) : Outer
        data class RemoveFavorite(val id: String) : Outer
        data object RetryLoad : Outer
    }

    sealed interface Inner : Msg {

        data class AddItem(val item: FavoriteItem) : Inner
        data class ItemLoadingResult(
            val result: Result<List<FavoriteItem>>
        ) : Inner
        sealed interface ItemRemoveResult : Inner {...}
    }
}
```



Msg

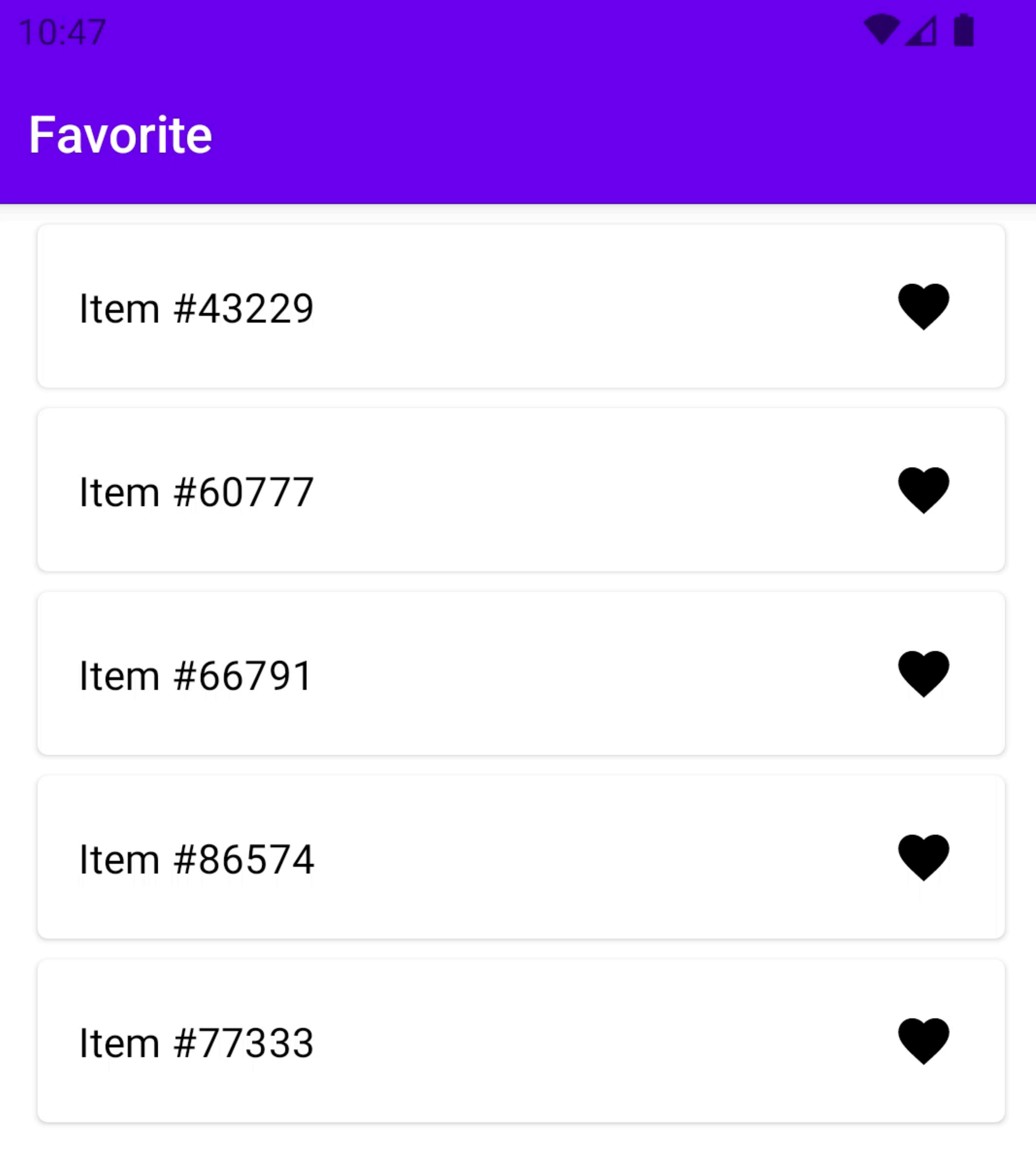
```kotlin
sealed interface Msg {

    sealed interface Outer : Msg {
        data class ItemClick(val id: String) : Outer
        data class RemoveFavorite(val id: String) : Outer
        data object RetryLoad : Outer
    }

    sealed interface Inner : Msg {

        data class AddItem(val item: FavoriteItem) : Inner
        data class ItemLoadingResult(
            val result: Result<List<FavoriteItem>>
        ) : Inner
        sealed interface ItemRemoveResult : Inner {...}
    }
}
```



Added item #18995

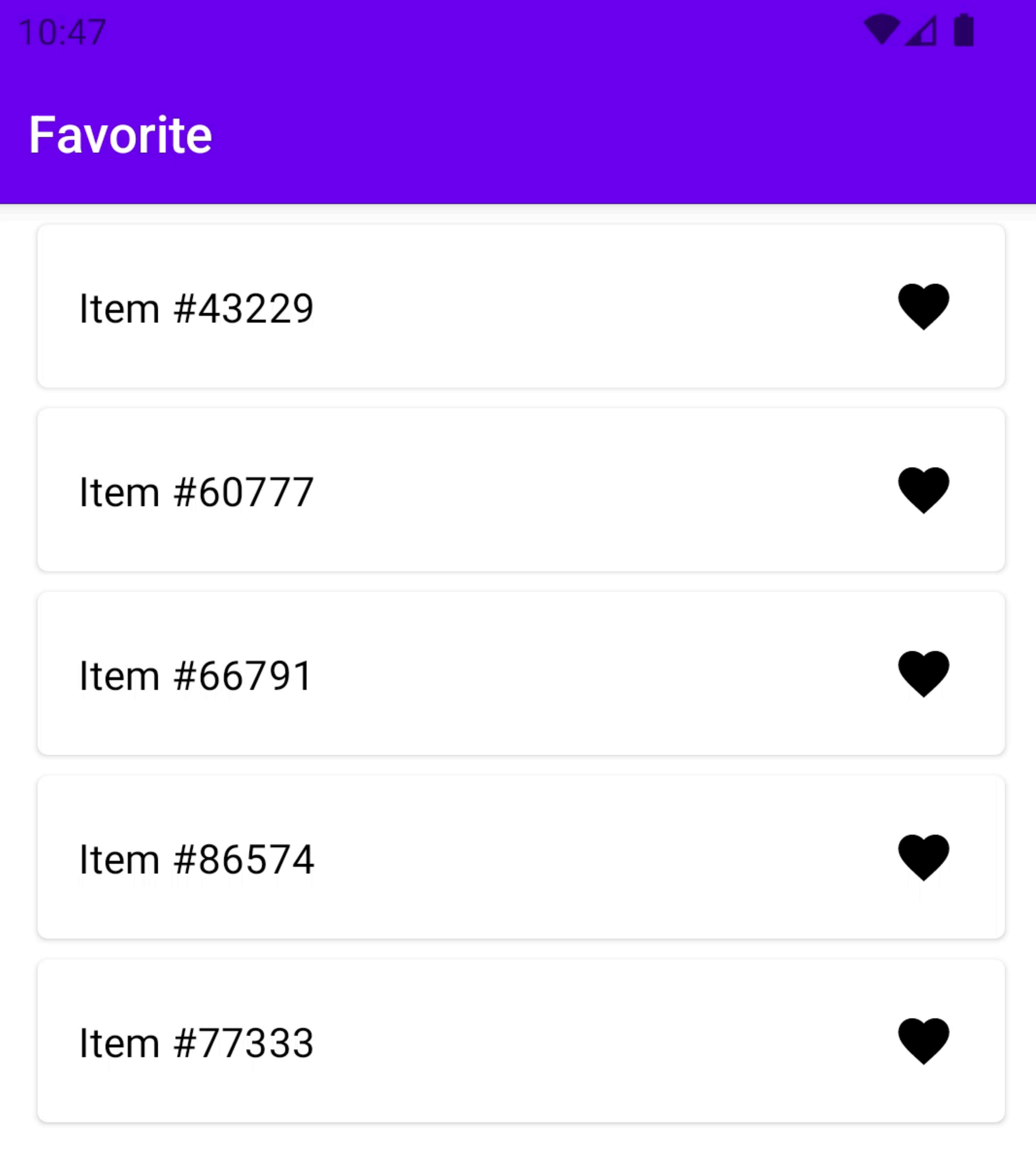
Msg

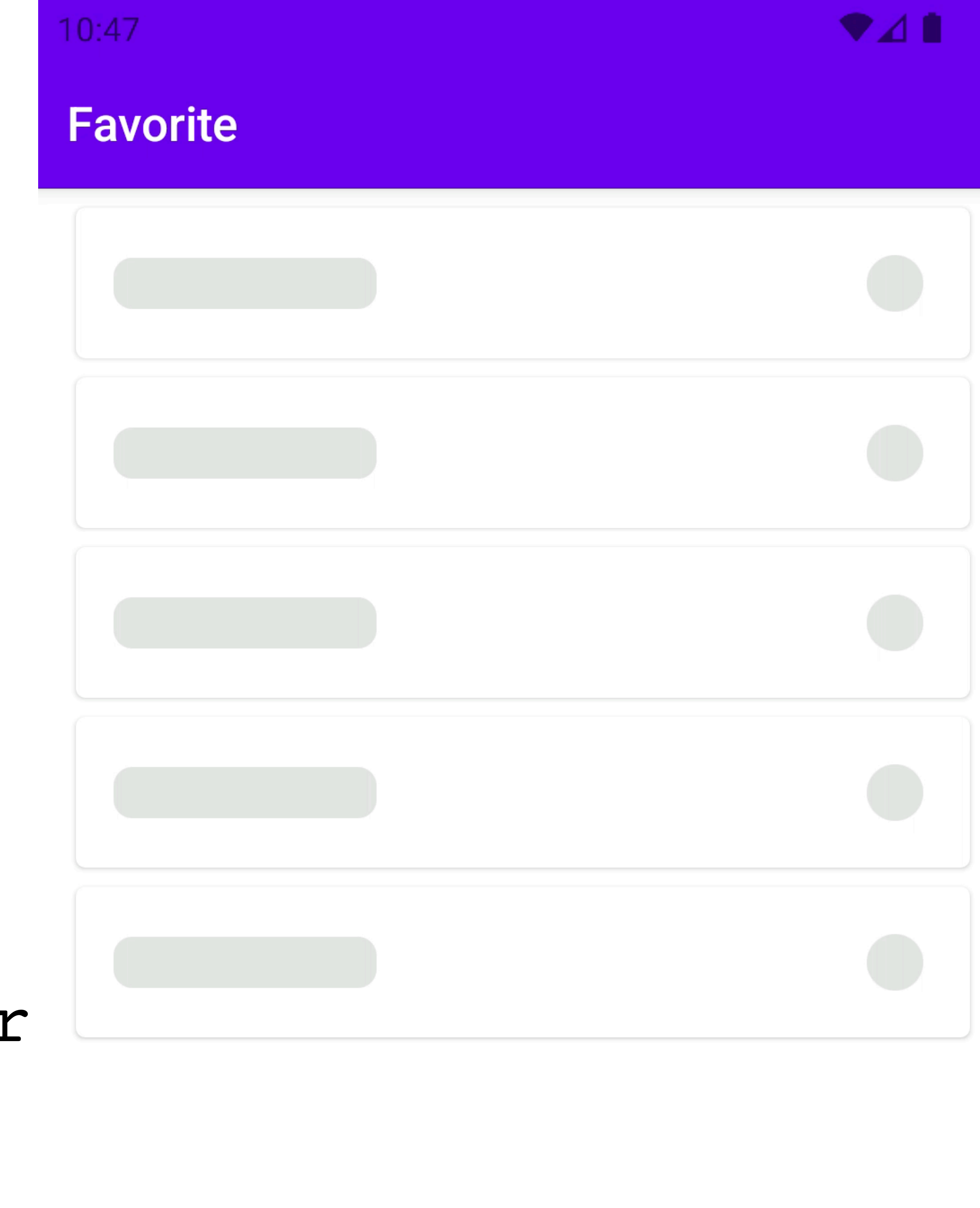
```
sealed interface Msg {

    sealed interface Outer : Msg {
        data class ItemClick(val id: String) : Outer
        data class RemoveFavorite(val id: String) : Outer
        data object RetryLoad : Outer
    }

    sealed interface Inner : Msg {

        data class AddItem(val item: FavoriteItem) : Inner
        data class ItemLoadingResult(
            val result: Result<List<FavoriteItem>>
        ) : Inner
        sealed interface ItemRemoveResult : Inner {...}
    }
}
```



Msg

```kotlin
sealed interface Msg {

    sealed interface Outer : Msg {
        data class ItemClick(val id: String) : Outer
        data class RemoveFavorite(val id: String) : Outer
        data object RetryLoad : Outer
    }

    sealed interface Inner : Msg {

        data class AddItem(val item: FavoriteItem) : Inner
        data class ItemLoadingResult(
            val result: Result<List<FavoriteItem>>
        ) : Inner
        sealed interface ItemRemoveResult : Inner {...}
    }
}
```

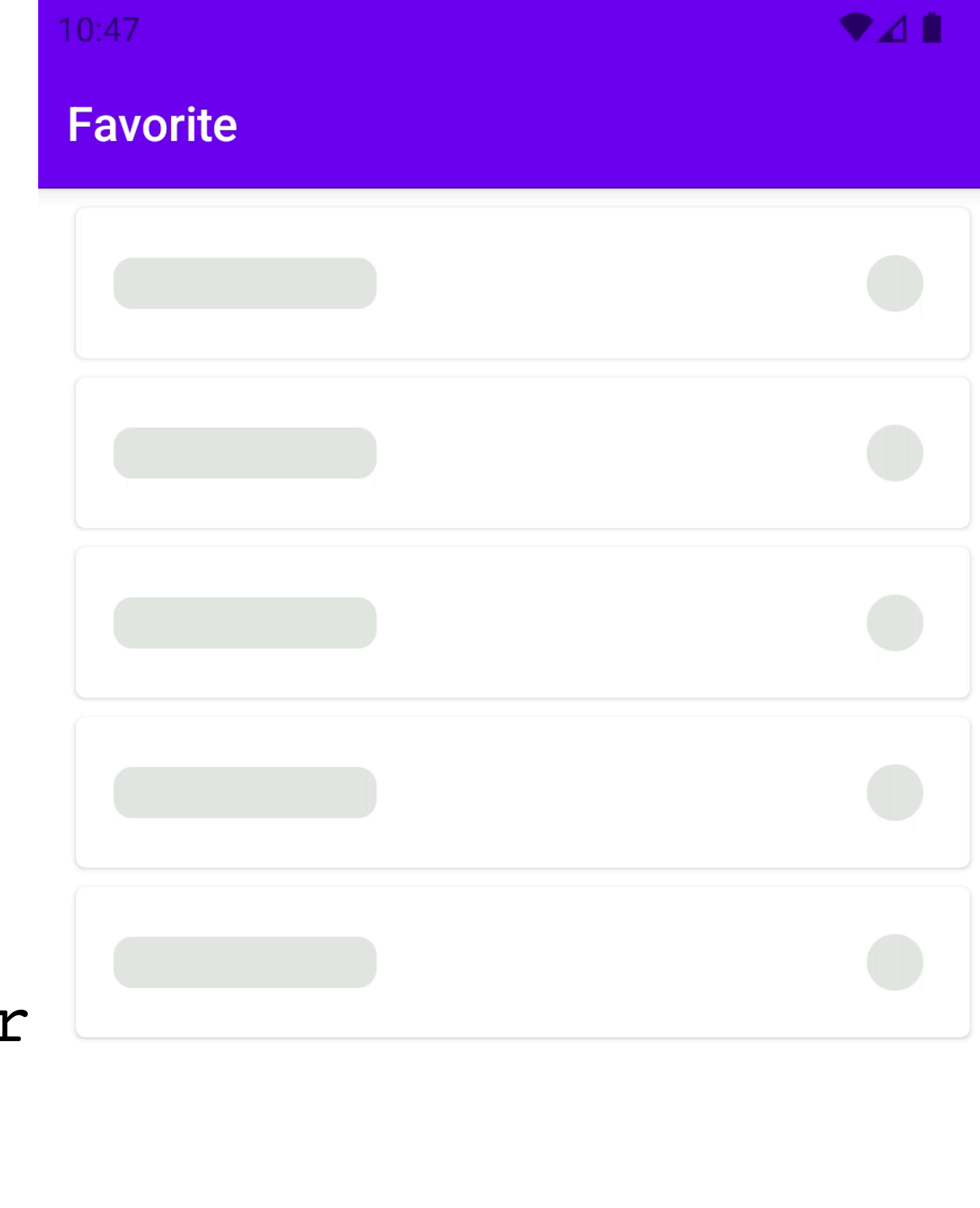





```kotlin
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```

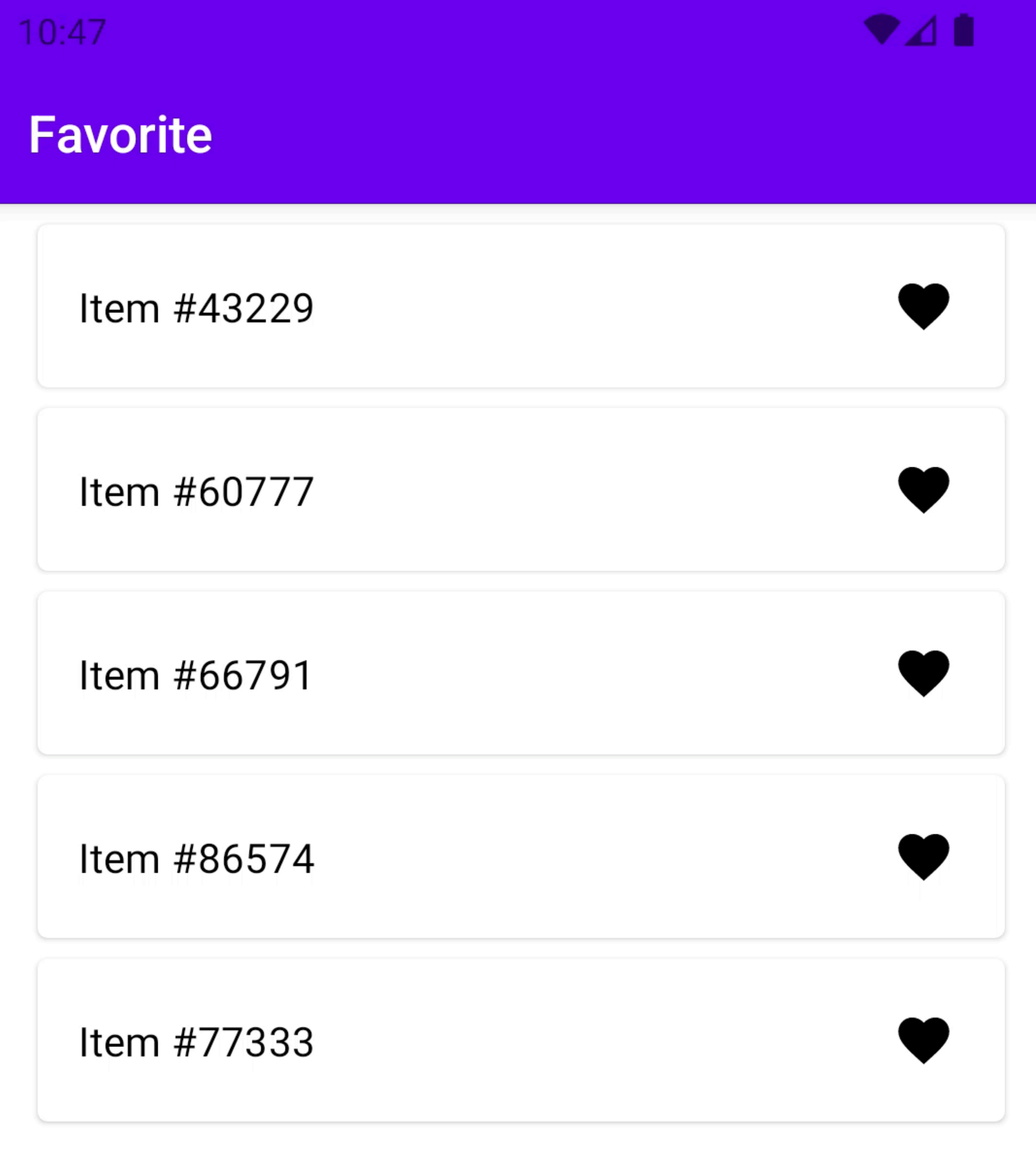
Eff

```kotlin
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```



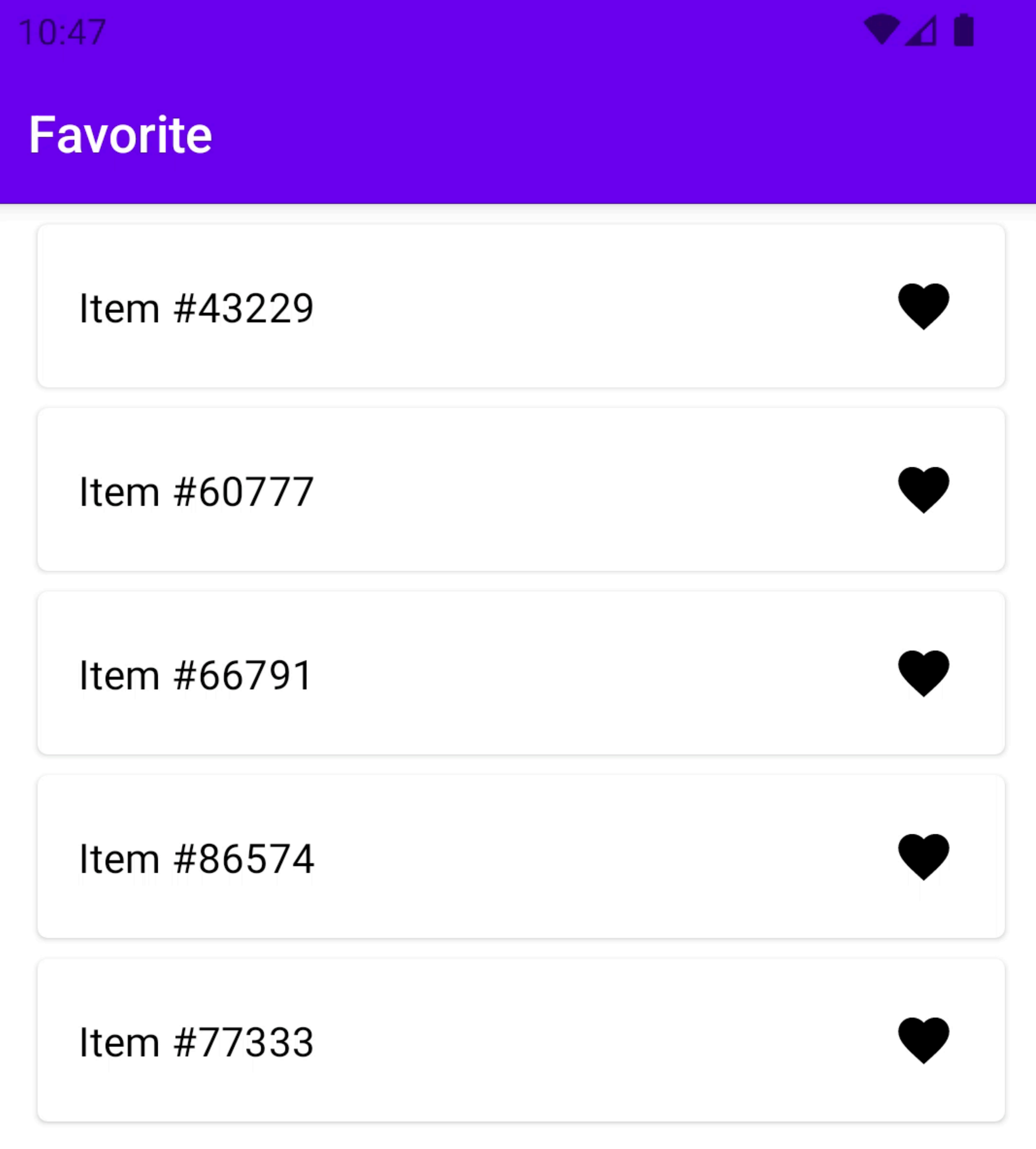
Eff

```
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```



Eff

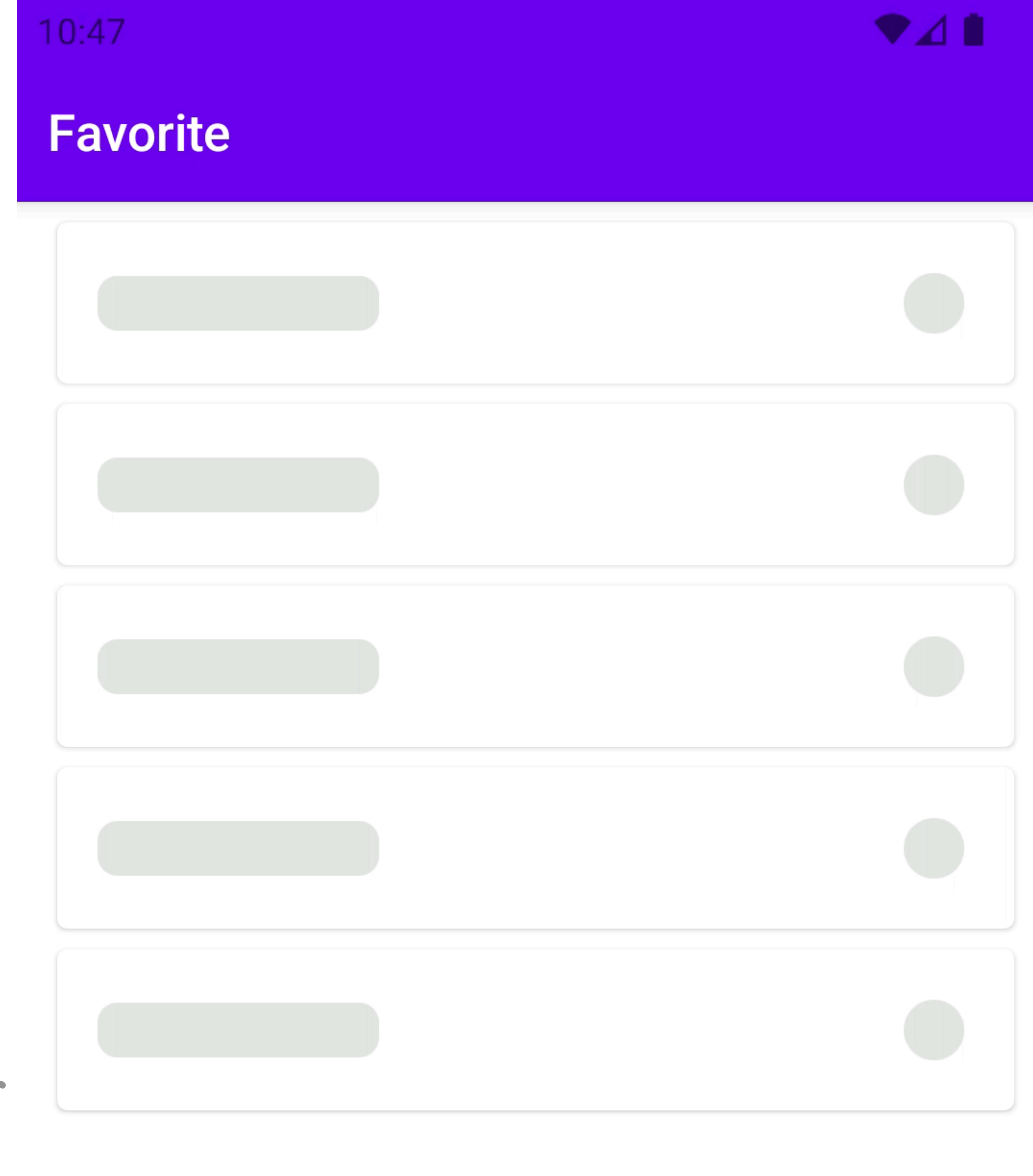
```
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```



Eff

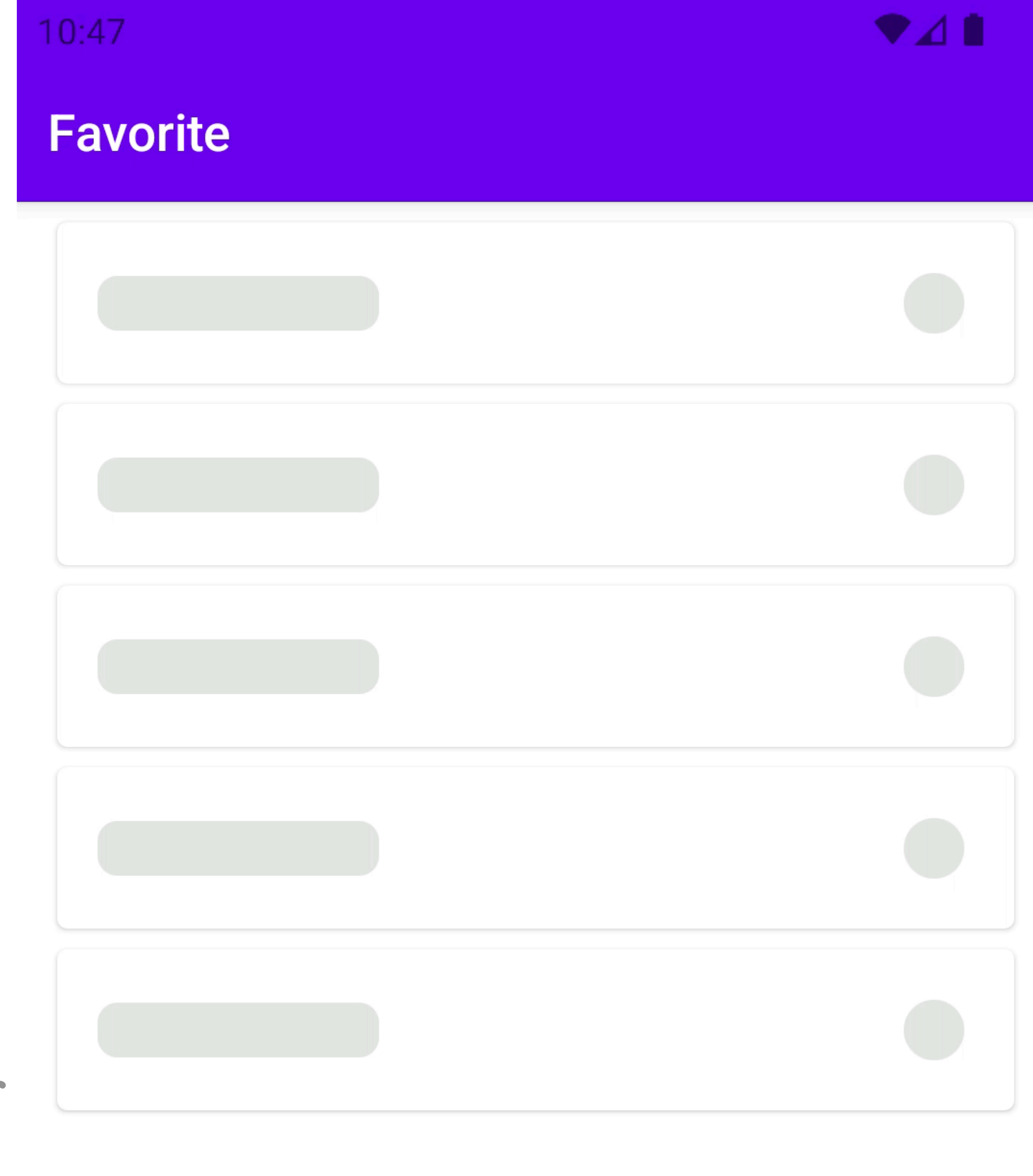
```
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```

Eff




```kotlin
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```

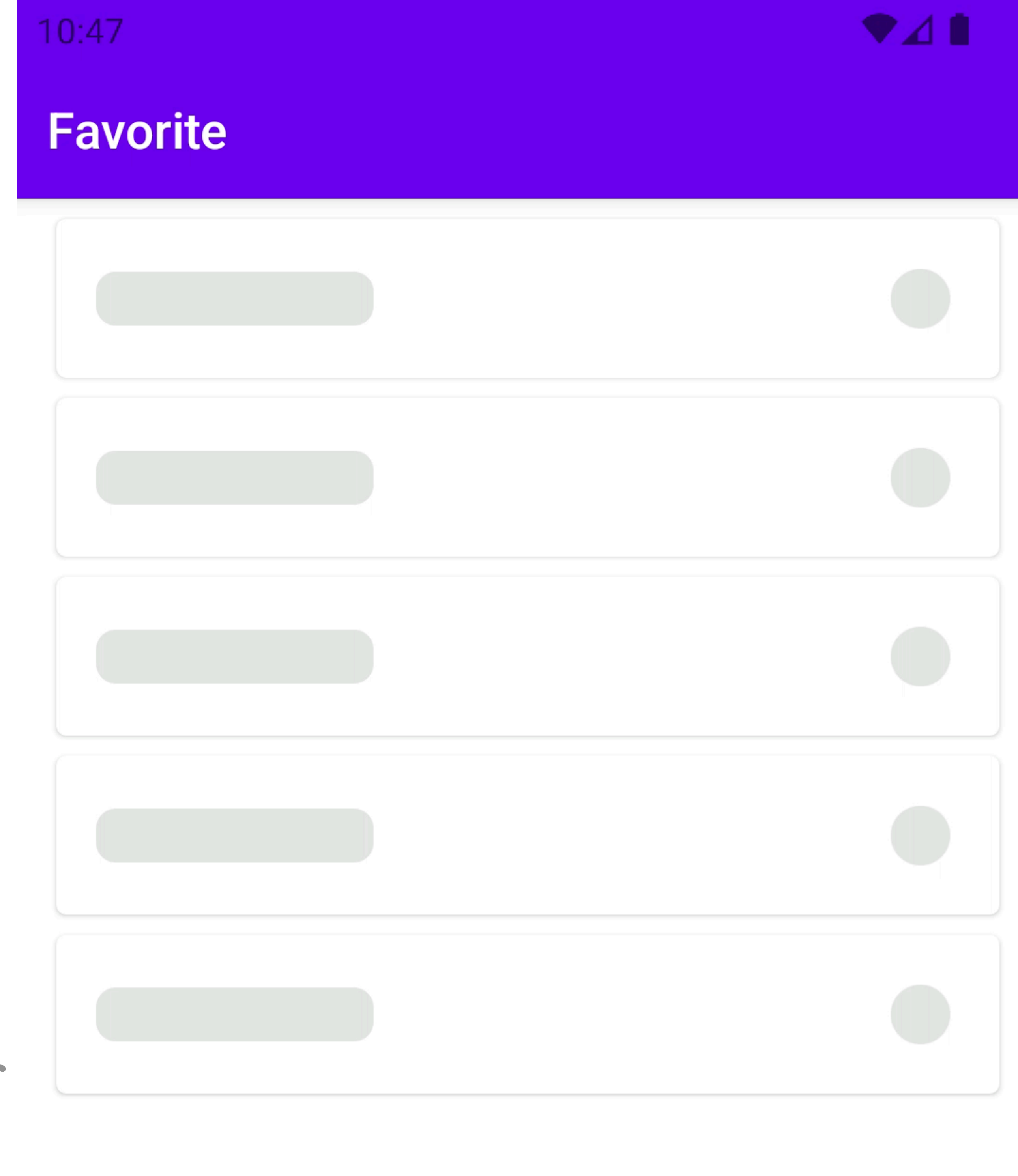
Eff




```
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```

Eff

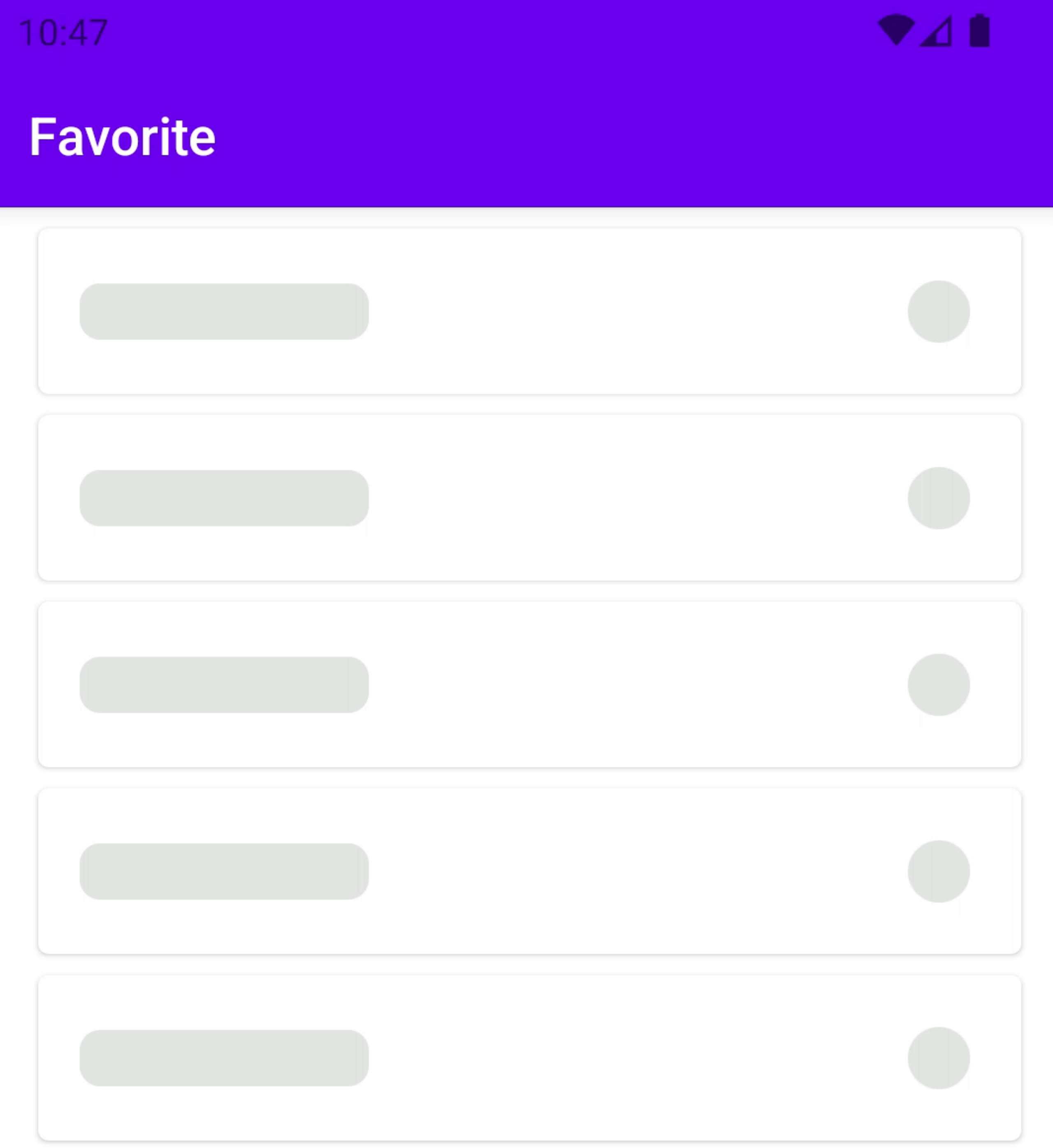
```
sealed interface Eff {

    sealed interface Outer : Eff {
        data class ItemAdded(val id: String) : Outer
        data class ItemRemoved(val id: String) : Outer
        data class ItemRemoveError(val id: String) : Outer
        data class ItemClick(val id: String) : Outer
    }

    sealed interface Inner : Eff {
        data object LoadFav : Inner
        data class RemoveItem(val id: String) : Inner
        data object ObserveFavUpdates : Inner
    }
}
```



Internal interaction

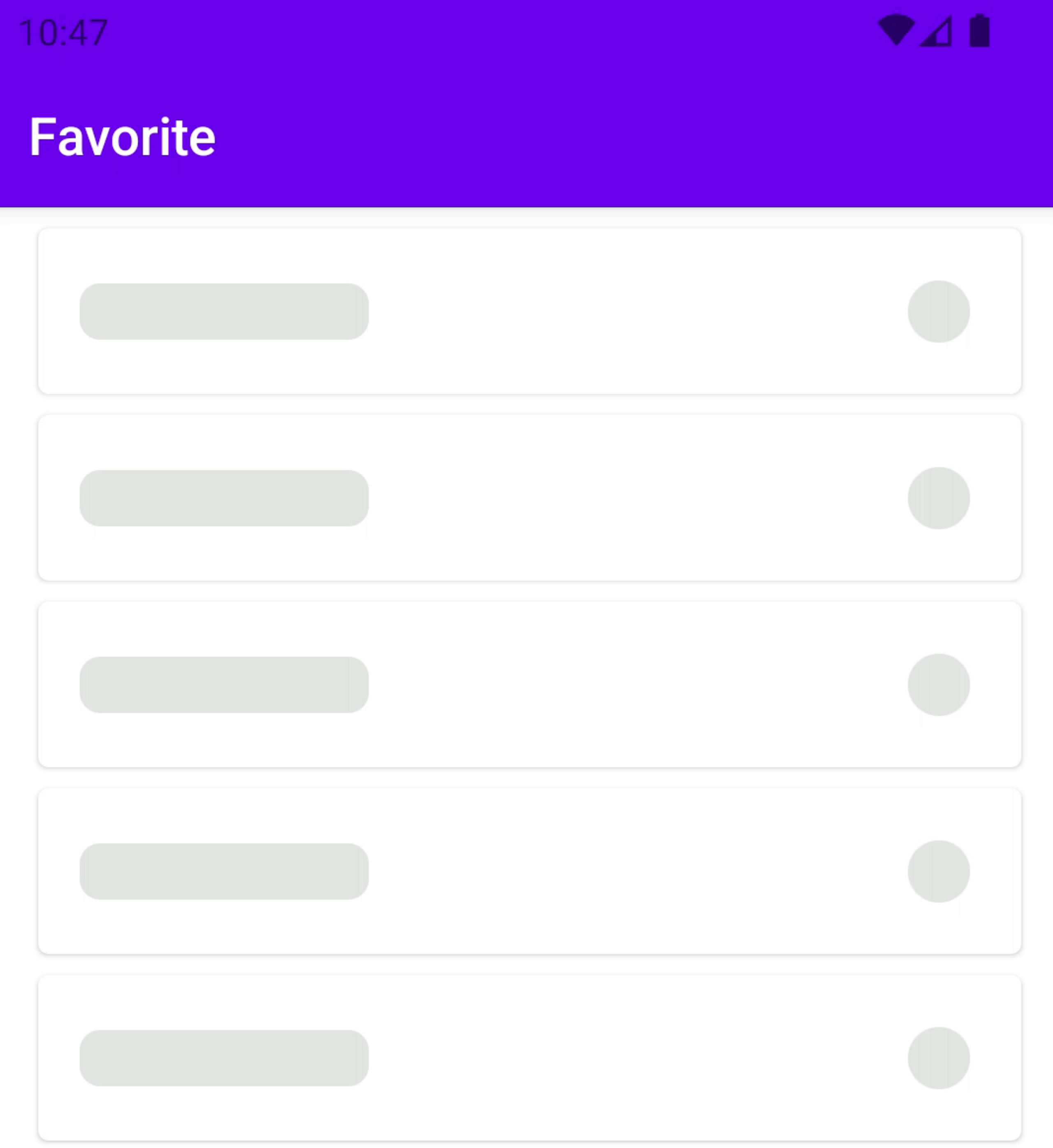
State

```
internal data class State(
    val content: LCE<List<FavoriteItem>>,
)

data class FavoriteItem(
    val id: String,
    val title: String,
    val isFavorite: Boolean = true,
    val updatingFavorite: Boolean = false
) : Serializable
```



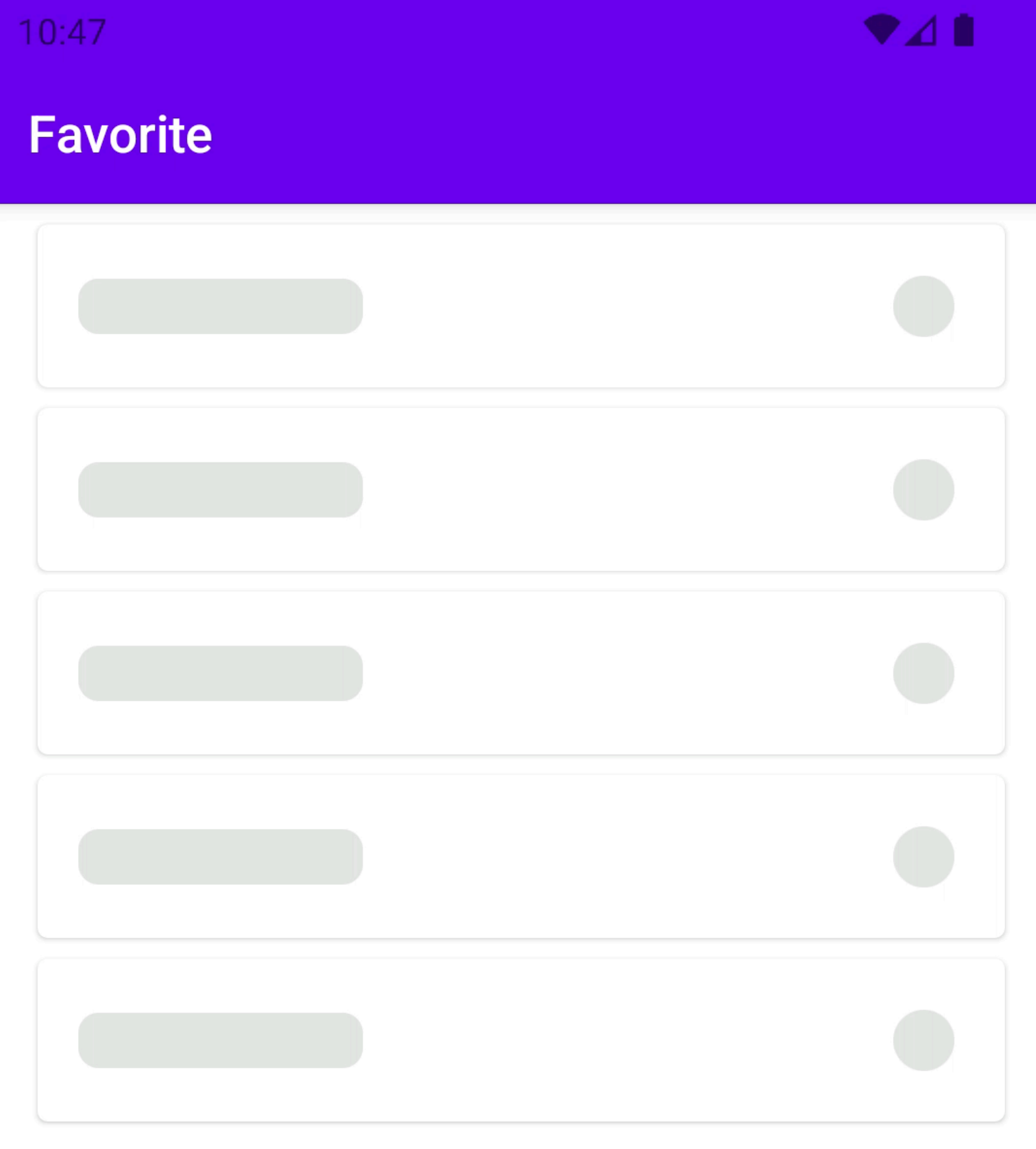
State

```
internal data class State(
    val content: LCE<List<FavoriteItem>>,
)

data class FavoriteItem(
    val id: String,
    val title: String,
    val isFavorite: Boolean = true,
    val updatingFavorite: Boolean = false
) : Serializable
```



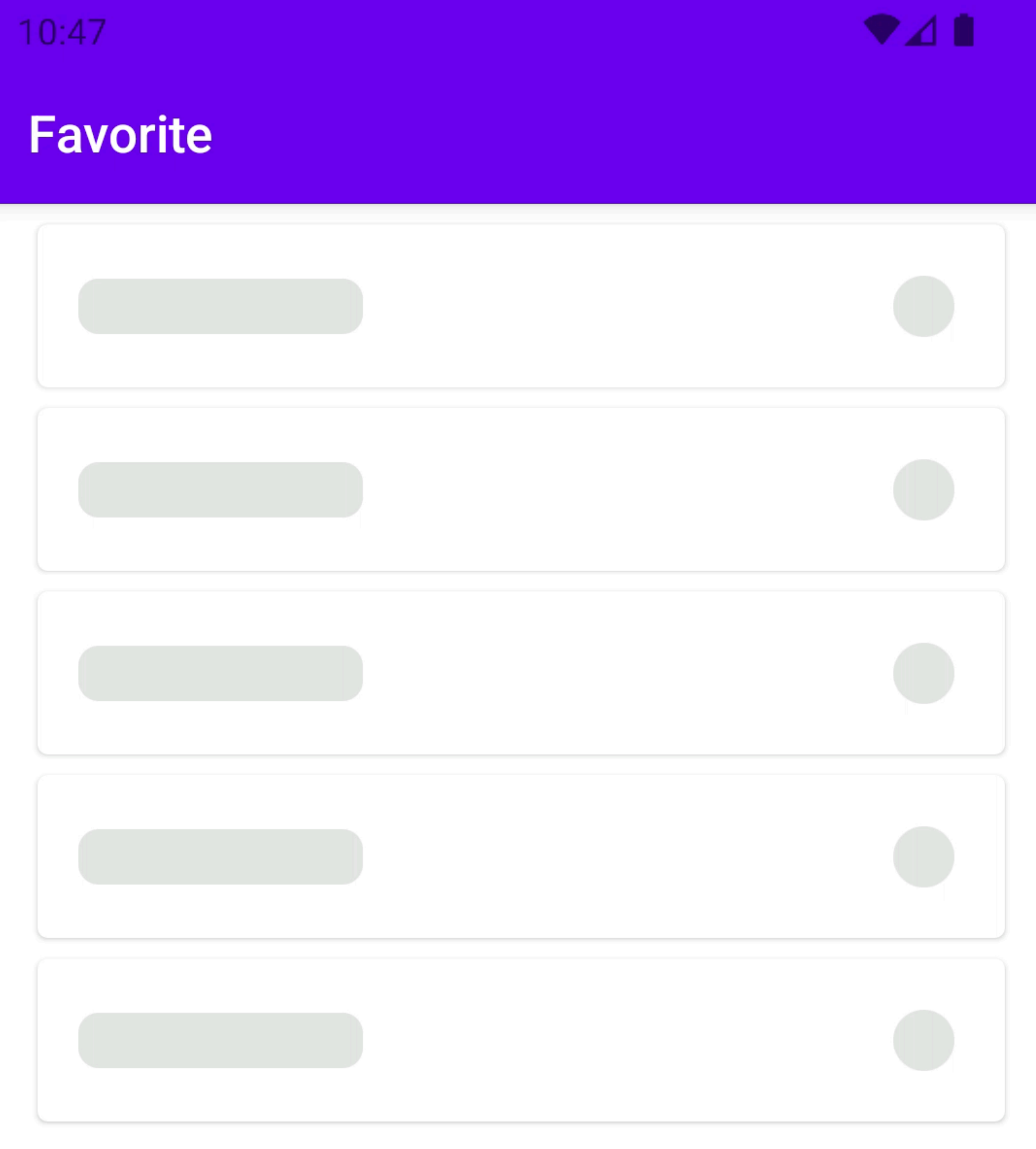
UiState

```kotlin
@Immutable
internal data class FavoriteUiState(
    val listCells: LCE<List<FavoriteListItem>>,
)

sealed interface FavoriteListItem : Parcelable {

    @Parcelize
    data class Item(
        val id: String,
        val title: String
    ) : FavoriteListItem

    @Parcelize
    data class Skeleton(val pos: Int) : FavoriteListItem

}
```



UiState

```kotlin
@Immutable
internal data class FavoriteUiState(
    val listCells: LCE<List<FavoriteListItem>>,
)

sealed interface FavoriteListItem : Parcelable {

    @Parcelize
    data class Item(
        val id: String,
        val title: String
    ) : FavoriteListItem

    @Parcelize
    data class Skeleton(val pos: Int) : FavoriteListItem

}
```



UiStore - convert models

```
val uiStore: Store<Msg.Outer, FavoriteUiState, Eff.Outer> =
    store.uiBuilder()
        .using<Msg.Outer, FavoriteUiState, Eff.Outer>(
            uiStateConverter = { state -> FavoriteUiConverter.convert(state) },
        )

val uiStoreVerbose: Store<Msg.Outer, FavoriteUiState, Eff.Outer> =
    store.uiBuilder()
        .using<Msg.Outer, FavoriteUiState, Eff.Outer>(
            uiStateConverter = { state -> FavoriteUiConverter.convert(state) },
            uiMsgToMsgConverter = { it as Msg},
            uiEffConverter = { it as? Eff.Outer}
        )
```

UiStore - convert models

```
val uiStore: Store<Msg.Outer, FavoriteUiState, Eff.Outer> =
    store.uiBuilder()
        .using<Msg.Outer, FavoriteUiState, Eff.Outer>(
            uiStateConverter = { state -> FavoriteUiConverter.convert(state) },
        )

val uiStoreVerbose: Store<Msg.Outer, FavoriteUiState, Eff.Outer> =
    store.uiBuilder()
        .using<Msg.Outer, FavoriteUiState, Eff.Outer>(
            uiStateConverter = { state -> FavoriteUiConverter.convert(state) },
            uiMsgToMsgConverter = { it as Msg},
            uiEffConverter = { it as? Eff.Outer}
        )
```

UI store - features

```kotlin
fun <UiMsg : Any, UiState : Any, UiEff : Any> using(
    uiMsgToMsgConverter: (UiMsg) -> Msg,
    uiStateConverter: (State) -> UiState,
    uiEffConverter: (Eff) -> UiEff?,
    cacheUiEffects: Boolean = true,
    propagateCloseToOriginal: Boolean = true,
    uiDispatcher: CoroutineDispatcher = Dispatchers.Main,
): UiStore<UiMsg, UiState, UiEff, Msg, State, Eff>
```

UI store - features

```
fun <UiMsg : Any, UiState : Any, UiEff : Any> using(
    uiMsgToMsgConverter: (UiMsg) -> Msg,
    uiStateConverter: (State) -> UiState,
    uiEffConverter: (Eff) -> UiEff?,
    cacheUiEffects: Boolean = true,
    propagateCloseToOriginal: Boolean = true,
    uiDispatcher: CoroutineDispatcher = Dispatchers.Main,
): UiStore<UiMsg, UiState, UiEff, Msg, State, Eff>
```

UI store - features

```
fun <UiMsg : Any, UiState : Any, UiEff : Any> using(
    uiMsgToMsgConverter: (UiMsg) -> Msg,
    uiStateConverter: (State) -> UiState,
    uiEffConverter: (Eff) -> UiEff?,
    cacheUiEffects: Boolean = true,
    propagateCloseToOriginal: Boolean = true,
    uiDispatcher: CoroutineDispatcher = Dispatchers.Main,
): UiStore<UiMsg, UiState, UiEff, Msg, State, Eff>
```

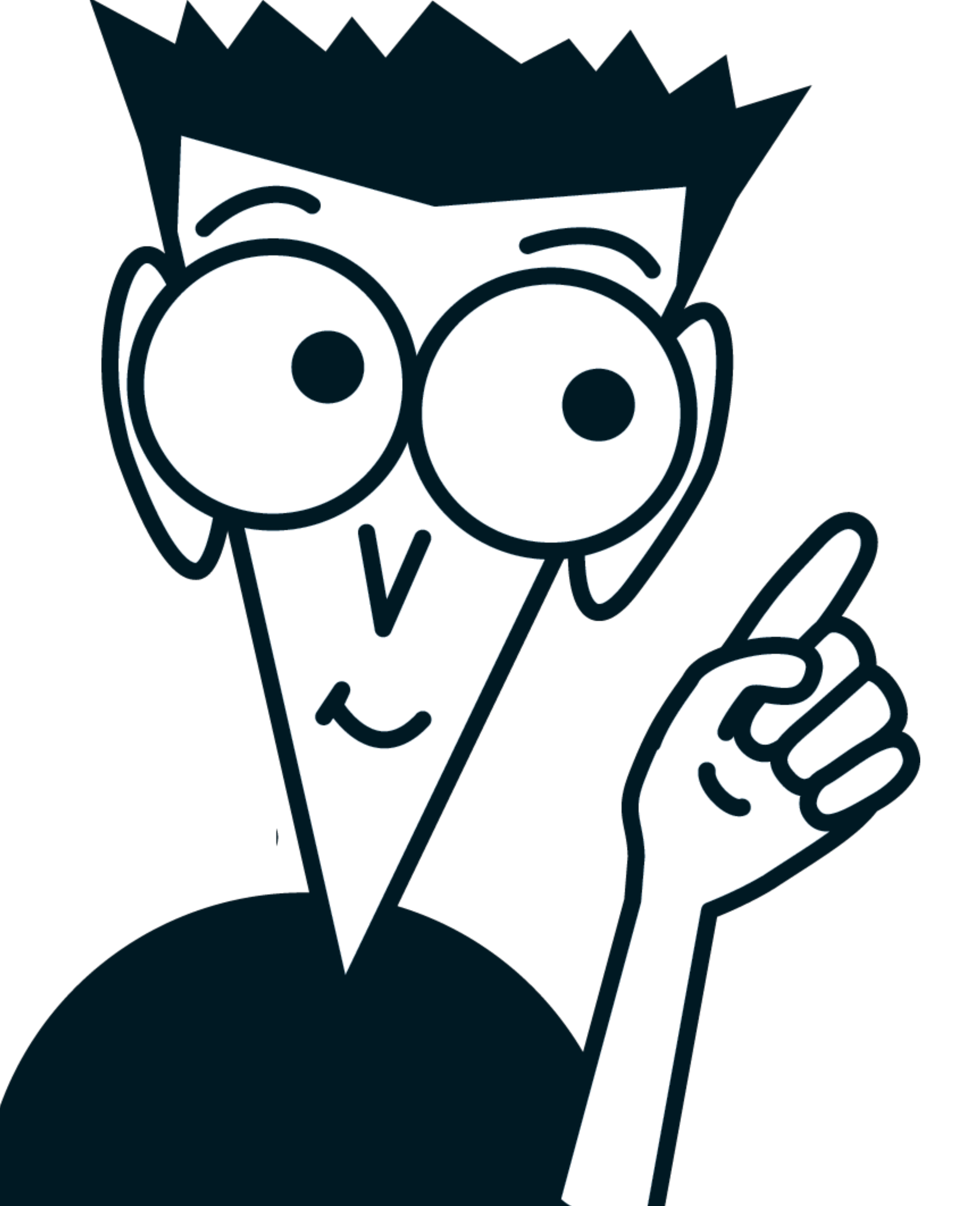
UI store - features

```
fun <UiMsg : Any, UiState : Any, UiEff : Any> using(
    uiMsgToMsgConverter: (UiMsg) -> Msg,
    uiStateConverter: (State) -> UiState,
    uiEffConverter: (Eff) -> UiEff?,
    cacheUiEffects: Boolean = true,
    propagateCloseToOriginal: Boolean = true,
    uiDispatcher: CoroutineDispatcher = Dispatchers.Main,
): UiStore<UiMsg, UiState, UiEff, Msg, State, Eff>
```

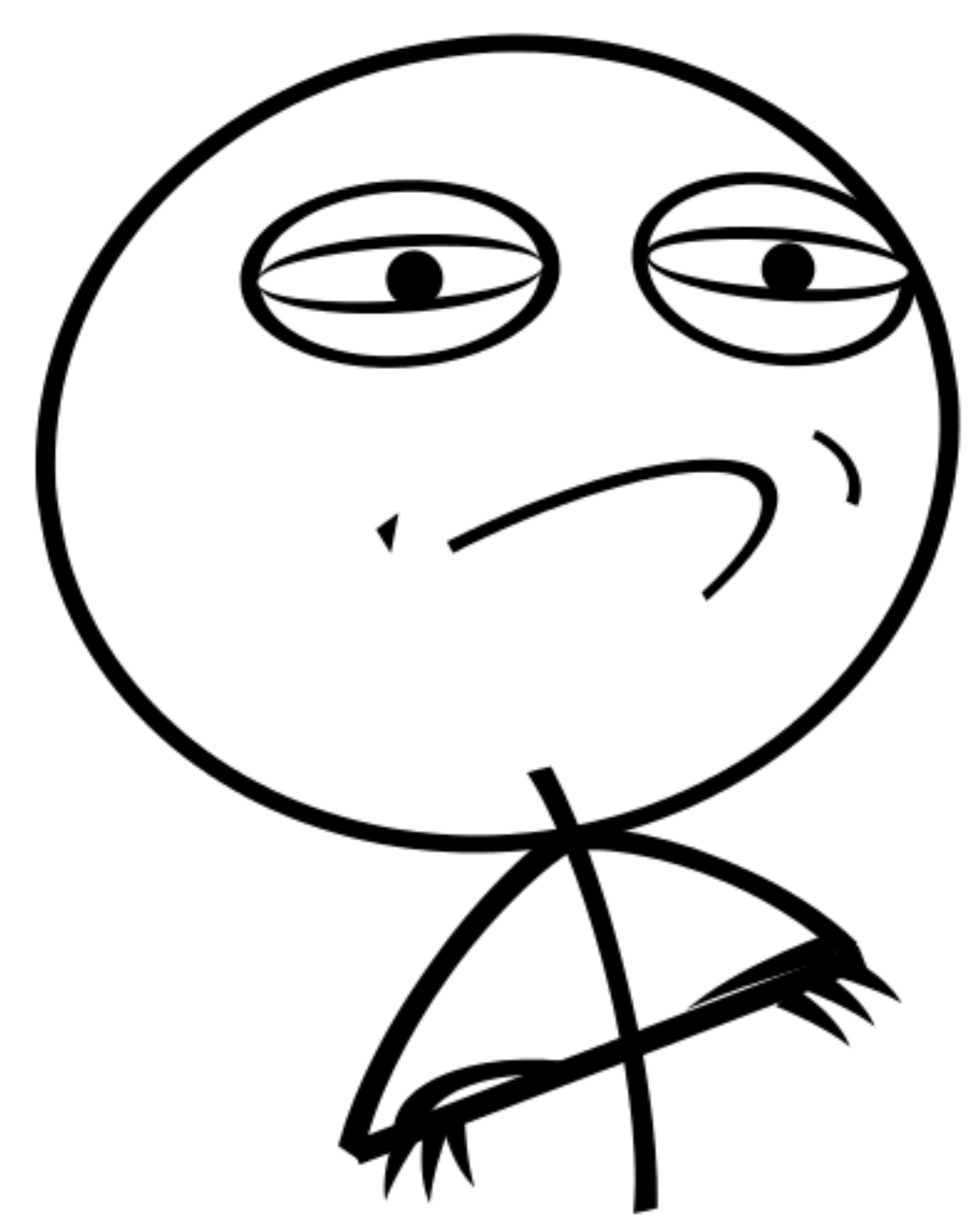
UI store - features

```
fun <UiMsg : Any, UiState : Any, UiEff : Any> using(
    uiMsgToMsgConverter: (UiMsg) -> Msg,
    uiStateConverter: (State) -> UiState,
    uiEffConverter: (Eff) -> UiEff?,
    cacheUiEffects: Boolean = true,
    propagateCloseToOriginal: Boolean = true,
    uiDispatcher: CoroutineDispatcher = Dispatchers.Main,
): UiStore<UiMsg, UiState, UiEff, Msg, State, Eff>
```

Do we need **VM for it?**



Yes and No!



We rather need **State keeping**

Instance Keeping in Compose

```
val uiStore: Store<Msg, FavoriteFeature.State, Eff> = rememberKombuchaStore {
    favoriteSampleFacade.scope.get<FavoriteStore>()
}
```

Instance Keeping in Compose

```
private class ModoKombuchaScreenModel<UiMsg : Any, UiState : Any, UiEff : Any>(
    val store: Store<UiMsg, UiState, UiEff>
) : ScreenModel {

    override fun onDispose() {
        store.close()
    }

}

@Composable
fun <Msg : Any, State : Any, Eff : Any> Screen.rememberKombuchaStore(
    createStore: () -> Store<Msg, State, Eff>
): Store<Msg, State, Eff> = rememberScreenModel {
    ModoKombuchaScreenModel(createStore())
}.store
```

Reusing
EffectHandler

Favorite store - adapt EffectHandler

```
internal class FavoriteStore(
    effectHandler: KombuchaFavEffHandler,
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteFeature.reducer,
    initialState = State(LCE.Loading()),
    initialEffects = setOf(Eff.Inner.LoadFav, Eff.Inner.ObserveFavUpdates),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

Adapt EffectHandler - adaptCast

```
inline fun <reified Eff1 : Any, Msg1 : Any, Eff2 : Any, reified Msg2 : Any>
        EffectHandler<Eff1, Msg1>.adaptCast(): EffectHandler<Eff2, Msg2> =
    adapt(
        effAdapter = { it as? Eff1 },
        msgAdapter = { it as? Msg2 }
    )

fun <Eff1 : Any, Msg1 : Any, Eff2 : Any, Msg2 : Any> EffectHandler<Eff1, Msg1>.adapt(
    effAdapter: (Eff2) -> Eff1?,
    msgAdapter: (Msg1) -> Msg2? = { null }
): EffectHandler<Eff2, Msg2> = object : EffectHandler<Eff2, Msg2> {

    override fun handleEff(eff: Eff2): Flow<Msg2> = effAdapter(eff)
        ?.let {
            handleEff(eff = it).mapNotNull { msgAdapter(it) }
        }
        ?: emptyFlow()

}
```

Adapt EffectHandler - adaptCast

```
inline fun <reified Eff1 : Any, Msg1 : Any, Eff2 : Any, reified Msg2 : Any>
        EffectHandler<Eff1, Msg1>.adaptCast(): EffectHandler<Eff2, Msg2> =
    adapt(
        effAdapter = { it as? Eff1 },
        msgAdapter = { it as? Msg2 }
    )

fun <Eff1 : Any, Msg1 : Any, Eff2 : Any, Msg2 : Any> EffectHandler<Eff1, Msg1>.adapt(
    effAdapter: (Eff2) -> Eff1?,
    msgAdapter: (Msg1) -> Msg2? = { null }
): EffectHandler<Eff2, Msg2> = object : EffectHandler<Eff2, Msg2> {

    override fun handleEff(eff: Eff2): Flow<Msg2> = effAdapter(eff)
        ?.let {
            handleEff(eff = it).mapNotNull { msgAdapter(it) }
        }
        ?: emptyFlow()

}
```

Adapt EffectHandler - adaptCast

```
inline fun <reified Eff1 : Any, Msg1 : Any, Eff2 : Any, reified Msg2 : Any>
        EffectHandler<Eff1, Msg1>.adaptCast(): EffectHandler<Eff2, Msg2> =
    adapt(
        effAdapter = { it as? Eff1 },
        msgAdapter = { it as? Msg2 }
    )

fun <Eff1 : Any, Msg1 : Any, Eff2 : Any, Msg2 : Any> EffectHandler<Eff1, Msg1>.adapt(
    effAdapter: (Eff2) -> Eff1?,
    msgAdapter: (Msg1) -> Msg2? = { null }
): EffectHandler<Eff2, Msg2> = object : EffectHandler<Eff2, Msg2> {

    override fun handleEff(eff: Eff2): Flow<Msg2> = effAdapter(eff)
        ?.let {
            handleEff(eff = it).mapNotNull { msgAdapter(it) }
        }
        ?: emptyFlow()

}
```

Adapt EffectHandler - adaptCast

```
inline fun <reified Eff1 : Any, Msg1 : Any, Eff2 : Any, reified Msg2 : Any>
        EffectHandler<Eff1, Msg1>.adaptCast(): EffectHandler<Eff2, Msg2> =
    adapt(
        effAdapter = { it as? Eff1 },
        msgAdapter = { it as? Msg2 }
    )

fun <Eff1 : Any, Msg1 : Any, Eff2 : Any, Msg2 : Any> EffectHandler<Eff1, Msg1>.adapt(
    effAdapter: (Eff2) -> Eff1?,
    msgAdapter: (Msg1) -> Msg2? = { null }
): EffectHandler<Eff2, Msg2> = object : EffectHandler<Eff2, Msg2> {

    override fun handleEff(eff: Eff2): Flow<Msg2> = effAdapter(eff)
        ?.let {
            handleEff(eff = it).mapNotNull { msgAdapter(it) }
        }
        ?: emptyFlow()

}
```

Adapt EffectHandler - adapt

```
inline fun <reified Eff1 : Any, Msg1 : Any, Eff2 : Any, reified Msg2 : Any>
        EffectHandler<Eff1, Msg1>.adaptCast(): EffectHandler<Eff2, Msg2> =
    adapt(
        effAdapter = { it as? Eff1 },
        msgAdapter = { it as? Msg2 }
    )

fun <Eff1 : Any, Msg1 : Any, Eff2 : Any, Msg2 : Any> EffectHandler<Eff1, Msg1>.adapt(
    effAdapter: (Eff2) -> Eff1?,
    msgAdapter: (Msg1) -> Msg2? = { null }
): EffectHandler<Eff2, Msg2> = object : EffectHandler<Eff2, Msg2> {

    override fun handleEff(eff: Eff2): Flow<Msg2> = effAdapter(eff)
        ?.let {
            handleEff(eff = it).mapNotNull { msgAdapter(it) }
        }
        ?: emptyFlow()

}
```

Adapt EffectHandler - adapt

```
inline fun <reified Eff1 : Any, Msg1 : Any, Eff2 : Any, reified Msg2 : Any>
        EffectHandler<Eff1, Msg1>.adaptCast(): EffectHandler<Eff2, Msg2> =
    adapt(
        effAdapter = { it as? Eff1 },
        msgAdapter = { it as? Msg2 }
    )

fun <Eff1 : Any, Msg1 : Any, Eff2 : Any, Msg2 : Any> EffectHandler<Eff1, Msg1>.adapt(
    effAdapter: (Eff2) -> Eff1?,
    msgAdapter: (Msg1) -> Msg2? = { null }
): EffectHandler<Eff2, Msg2> = object : EffectHandler<Eff2, Msg2> {

    override fun handleEff(eff: Eff2): Flow<Msg2> = effAdapter(eff)
        ?.let {
            handleEff(eff = it).mapNotNull { msgAdapter(it) }
        }
        ?: emptyFlow()

}
```

Adapt EffectHandler - adapt

```
inline fun <reified Eff1 : Any, Msg1 : Any, Eff2 : Any, reified Msg2 : Any>
        EffectHandler<Eff1, Msg1>.adaptCast(): EffectHandler<Eff2, Msg2> =
    adapt(
        effAdapter = { it as? Eff1 },
        msgAdapter = { it as? Msg2 }
    )

fun <Eff1 : Any, Msg1 : Any, Eff2 : Any, Msg2 : Any> EffectHandler<Eff1, Msg1>.adapt(
    effAdapter: (Eff2) -> Eff1?,
    msgAdapter: (Msg1) -> Msg2? = { null }
): EffectHandler<Eff2, Msg2> = object : EffectHandler<Eff2, Msg2> {

    override fun handleEff(eff: Eff2): Flow<Msg2> = effAdapter(eff)
        ?.let {
            handleEff(eff = it).mapNotNull { msgAdapter(it) }
        }
        ?: emptyFlow()

}
```

Adapt EffectHandler - adapt

```
inline fun <reified Eff1 : Any, Msg1 : Any, Eff2 : Any, reified Msg2 : Any>
        EffectHandler<Eff1, Msg1>.adaptCast(): EffectHandler<Eff2, Msg2> =
    adapt(
        effAdapter = { it as? Eff1 },
        msgAdapter = { it as? Msg2 }
    )

fun <Eff1 : Any, Msg1 : Any, Eff2 : Any, Msg2 : Any> EffectHandler<Eff1, Msg1>.adapt(
    effAdapter: (Eff2) -> Eff1?,
    msgAdapter: (Msg1) -> Msg2? = { null }
): EffectHandler<Eff2, Msg2> = object : EffectHandler<Eff2, Msg2> {

    override fun handleEff(eff: Eff2): Flow<Msg2> = effAdapter(eff)
        ?.let {
            handleEff(eff = it).mapNotNull { msgAdapter(it) }
        }
        ?: emptyFlow()

}
```

Adapt EffectHandler - adapt

```
inline fun <reified Eff1 : Any, Msg1 : Any, Eff2 : Any, reified Msg2 : Any>
        EffectHandler<Eff1, Msg1>.adaptCast(): EffectHandler<Eff2, Msg2> =
    adapt(
        effAdapter = { it as? Eff1 },
        msgAdapter = { it as? Msg2 }
    )

fun <Eff1 : Any, Msg1 : Any, Eff2 : Any, Msg2 : Any> EffectHandler<Eff1, Msg1>.adapt(
    effAdapter: (Eff2) -> Eff1?,
    msgAdapter: (Msg1) -> Msg2? = { null }
): EffectHandler<Eff2, Msg2> = object : EffectHandler<Eff2, Msg2> {

    override fun handleEff(eff: Eff2): Flow<Msg2> = effAdapter(eff)
        ?.let {
            handleEff(eff = it).mapNotNull { msgAdapter(it) }
        }
        ?: emptyFlow()

}
```

Adapt EffectHandler - adapt

```
inline fun <reified Eff1 : Any, Msg1 : Any, Eff2 : Any, reified Msg2 : Any>
        EffectHandler<Eff1, Msg1>.adaptCast(): EffectHandler<Eff2, Msg2> =
    adapt(
        effAdapter = { it as? Eff1 },
        msgAdapter = { it as? Msg2 }
    )

fun <Eff1 : Any, Msg1 : Any, Eff2 : Any, Msg2 : Any> EffectHandler<Eff1, Msg1>.adapt(
    effAdapter: (Eff2) -> Eff1?,
    msgAdapter: (Msg1) -> Msg2? = { null }
): EffectHandler<Eff2, Msg2> = object : EffectHandler<Eff2, Msg2> {

    override fun handleEff(eff: Eff2): Flow<Msg2> = effAdapter(eff)
        ?.let {
            handleEff(eff = it).mapNotNull { msgAdapter(it) }
        }
        ?: emptyFlow()

}
```

Adapt EffectHandler - adapt

```
inline fun <reified Eff1 : Any, Msg1 : Any, Eff2 : Any, reified Msg2 : Any>
        EffectHandler<Eff1, Msg1>.adaptCast(): EffectHandler<Eff2, Msg2> =
    adapt(
        effAdapter = { it as? Eff1 },
        msgAdapter = { it as? Msg2 }
    )

fun <Eff1 : Any, Msg1 : Any, Eff2 : Any, Msg2 : Any> EffectHandler<Eff1, Msg1>.adapt(
    effAdapter: (Eff2) -> Eff1?,
    msgAdapter: (Msg1) -> Msg2? = { null }
): EffectHandler<Eff2, Msg2> = object : EffectHandler<Eff2, Msg2> {

    override fun handleEff(eff: Eff2): Flow<Msg2> = effAdapter(eff)
        ?.let {
            handleEff(eff = it).mapNotNull { msgAdapter(it) }
        }
        ?: emptyFlow()

}
```

Adapt EffectHandler - adapt

```
inline fun <reified Eff1 : Any, Msg1 : Any, Eff2 : Any, reified Msg2 : Any>
        EffectHandler<Eff1, Msg1>.adaptCast(): EffectHandler<Eff2, Msg2> =
    adapt(
        effAdapter = { it as? Eff1 },
        msgAdapter = { it as? Msg2 }
    )

fun <Eff1 : Any, Msg1 : Any, Eff2 : Any, Msg2 : Any> EffectHandler<Eff1, Msg1>.adapt(
    effAdapter: (Eff2) -> Eff1?,
    msgAdapter: (Msg1) -> Msg2? = { null }
): EffectHandler<Eff2, Msg2> = object : EffectHandler<Eff2, Msg2> {

    override fun handleEff(eff: Eff2): Flow<Msg2> = effAdapter(eff)
        ?.let {
            handleEff(eff = it).mapNotNull { msgAdapter(it) }
        }
        ?: emptyFlow()

}
```

Adapt EffectHandler - adapt

```
inline fun <reified Eff1 : Any, Msg1 : Any, Eff2 : Any, reified Msg2 : Any>
        EffectHandler<Eff1, Msg1>.adaptCast(): EffectHandler<Eff2, Msg2> =
    adapt(
        effAdapter = { it as? Eff1 },
        msgAdapter = { it as? Msg2 }
    )

fun <Eff1 : Any, Msg1 : Any, Eff2 : Any, Msg2 : Any> EffectHandler<Eff1, Msg1>.adapt(
    effAdapter: (Eff2) -> Eff1?,
    msgAdapter: (Msg1) -> Msg2? = { null }
): EffectHandler<Eff2, Msg2> = object : EffectHandler<Eff2, Msg2> {

    override fun handleEff(eff: Eff2): Flow<Msg2> = effAdapter(eff)
        ?.let {
            handleEff(eff = it).mapNotNull { msgAdapter(it) }
        }
        ?: emptyFlow()

}
```

Favorite store - adapt EffectHandler

```
internal class FavoriteStore(
    effectHandler: KombuchaFavEffHandler,
) : CoroutinesStore<Msg, State, Eff>(
    name = "FavoriteStore",
    reducer = FavoriteFeature.reducer,
    initialState = State(LCE.Loading()),
    initialEffects = setOf(Eff.Inner.LoadFav, Eff.Inner.ObserveFavUpdates),
    effectHandlers = arrayOf(effectHandler.adaptCast())
)
```

Favorite store

```
internal class FavoriteEffectHandler(
    private val repository: FavoriteRepository,
) : EffectHandler<Eff.Inner, Msg.Inner>
```

Favorite store

```kotlin
override fun handleEff(eff: Eff.Inner): Flow<Msg.Inner> = when (eff) {
    is Eff.Inner.LoadFav -> flow {
        emit(
            Msg.Inner.ItemLoadingResult(
                runCatching { repository.loadFavoriteItems() }
            )
        )
    }
    is Eff.Inner.RemoveItem -> flow {
        emit(
            repository.removeFavoriteItem(eff.id).fold(
                onSuccess = { Msg.Inner.ItemRemoveResult.Done(eff.id) },
                onFailure = { Msg.Inner.ItemRemoveResult.Error(eff.id, null) }
            )
        )
    }
    Eff.Inner.ObserveFavUpdates -> repository.newFavoriteSource().map {
        Msg.Inner.AddItem(it)
    }
}
```

Stand-alone EffectHandler

```
class ChangeFavoriteEffectHandler: EffectHandler<ChangeFavoriteEff, ChangeFavoriteMsg>

data class ChangeFavoriteEff(val id: String, val favorite: Boolean)

sealed interface ChangeFavoriteMsg {
    val id: String
    val favorite: Boolean

    data class Done(
        override val id: String,
        override val favorite: Boolean
    ) : ChangeFavoriteMsg

    data class Error(
        override val id: String,
        override val favorite: Boolean,
        val throwable: Throwable?
    ) : ChangeFavoriteMsg
}
```

Stand-alone EffectHandler: adapt

```
val effHandler: EffectHandler<Eff, Msg> = ChangeFavoriteEffectHandler().adapt(
    effAdapter = { eff: Eff ->
        when (eff) {
            is Eff.Inner.RemoveItem -> ChangeFavoriteEff(eff.id, favorite = false)
            else -> null
        }
    },
    msgAdapter = { msg ->
        when (msg) {
            is ChangeFavoriteMsg.Done ->
                Msg.Inner.ItemRemoveResult.Done(msg.id)
            is ChangeFavoriteMsg.Error ->
                Msg.Inner.ItemRemoveResult.Error(msg.id, msg.throwable)
        }

    }
)
```

Migration to TEA
Tips & Tricks

01

Integrate existed code using bindings or effects

02

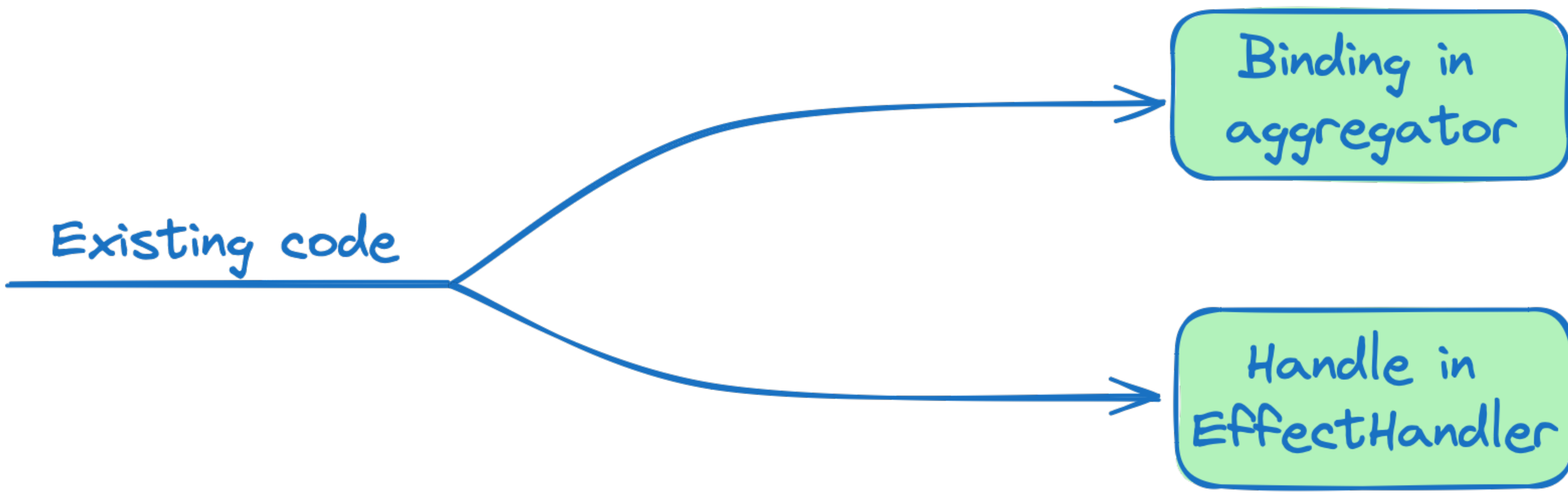
Migrate stateful components to store

Integrate existed code

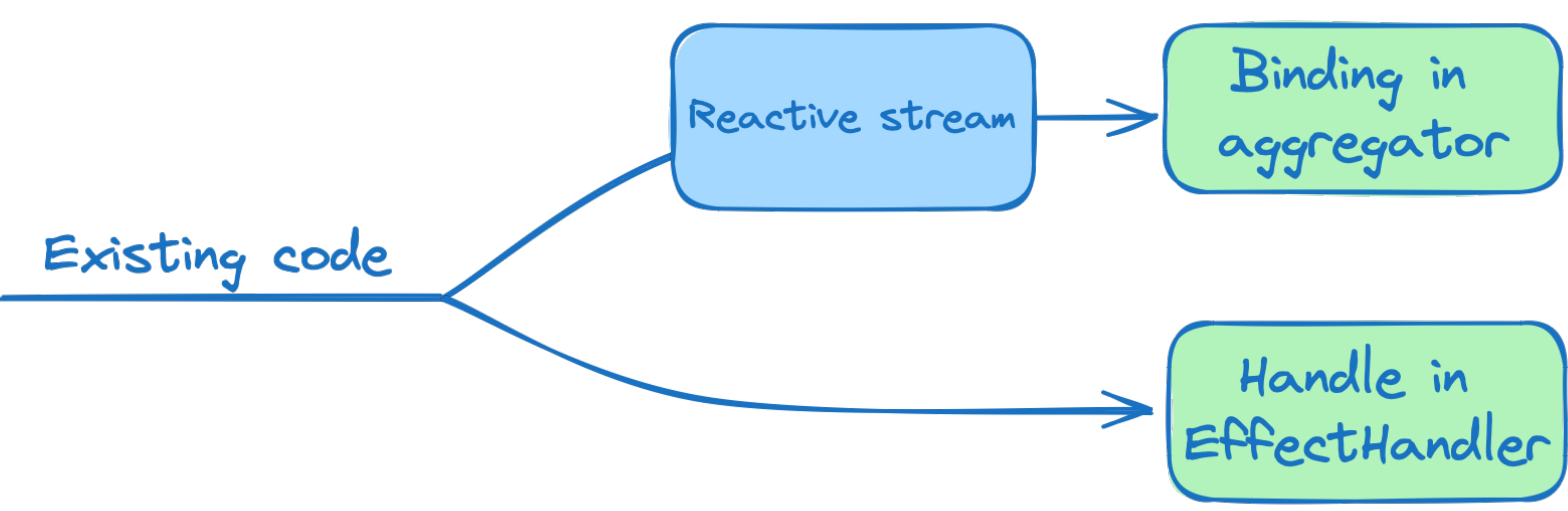
Existing code

Integrate existed code

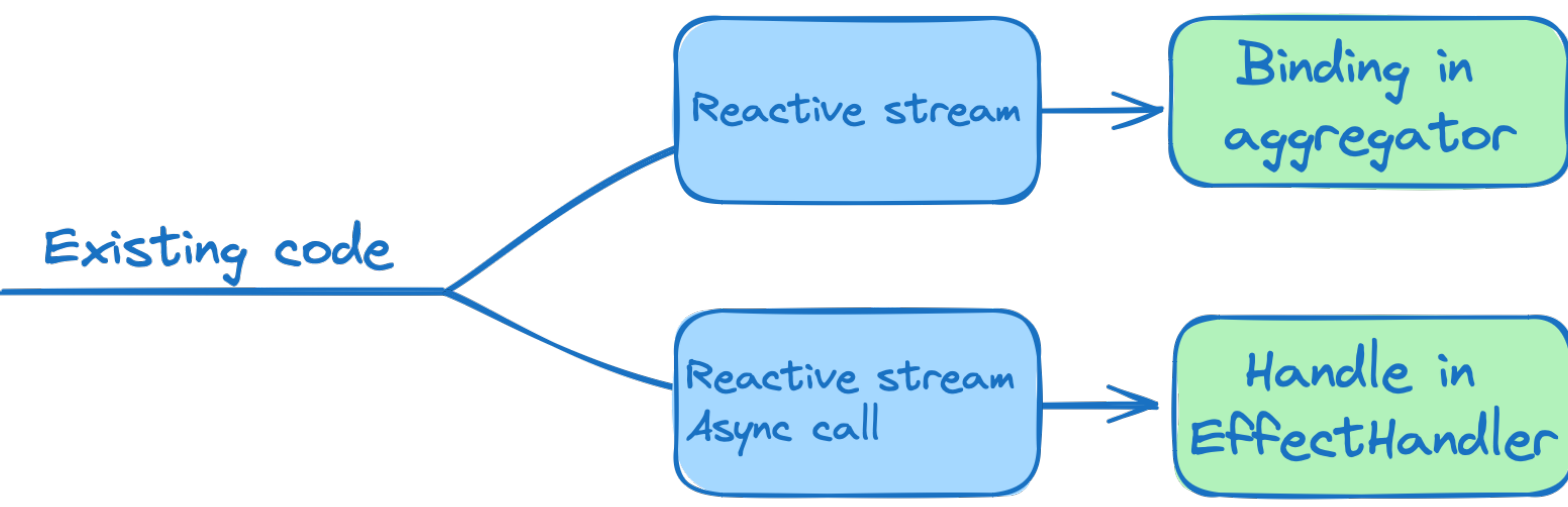
Integrate existed code



Integrate existed code

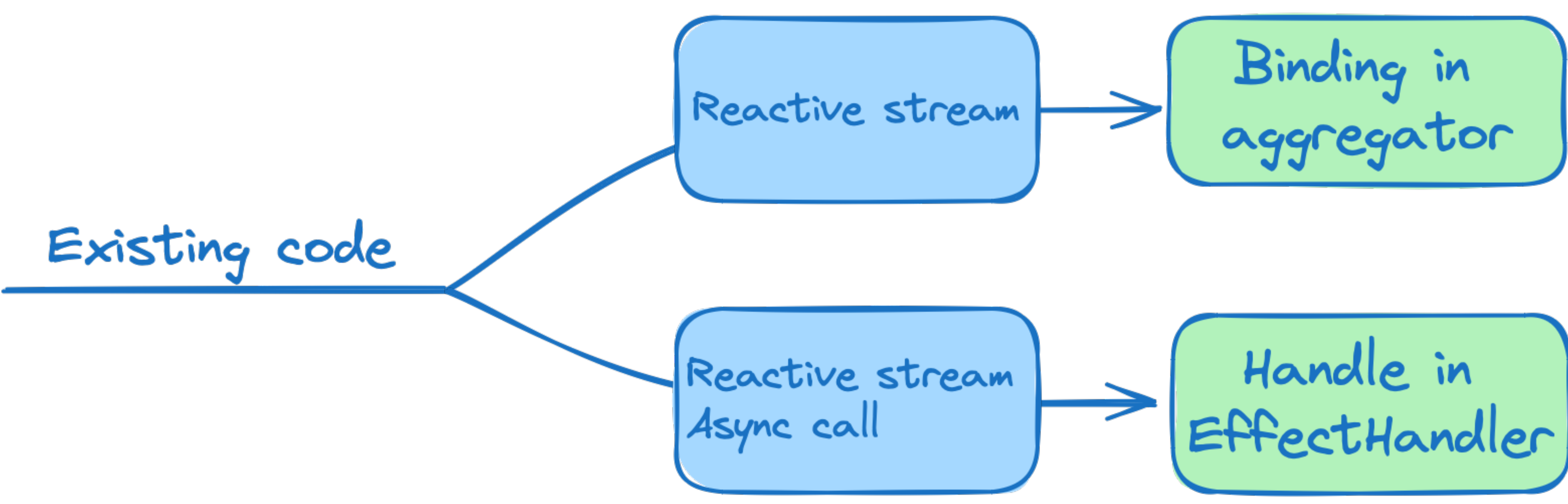


Integrate existed code



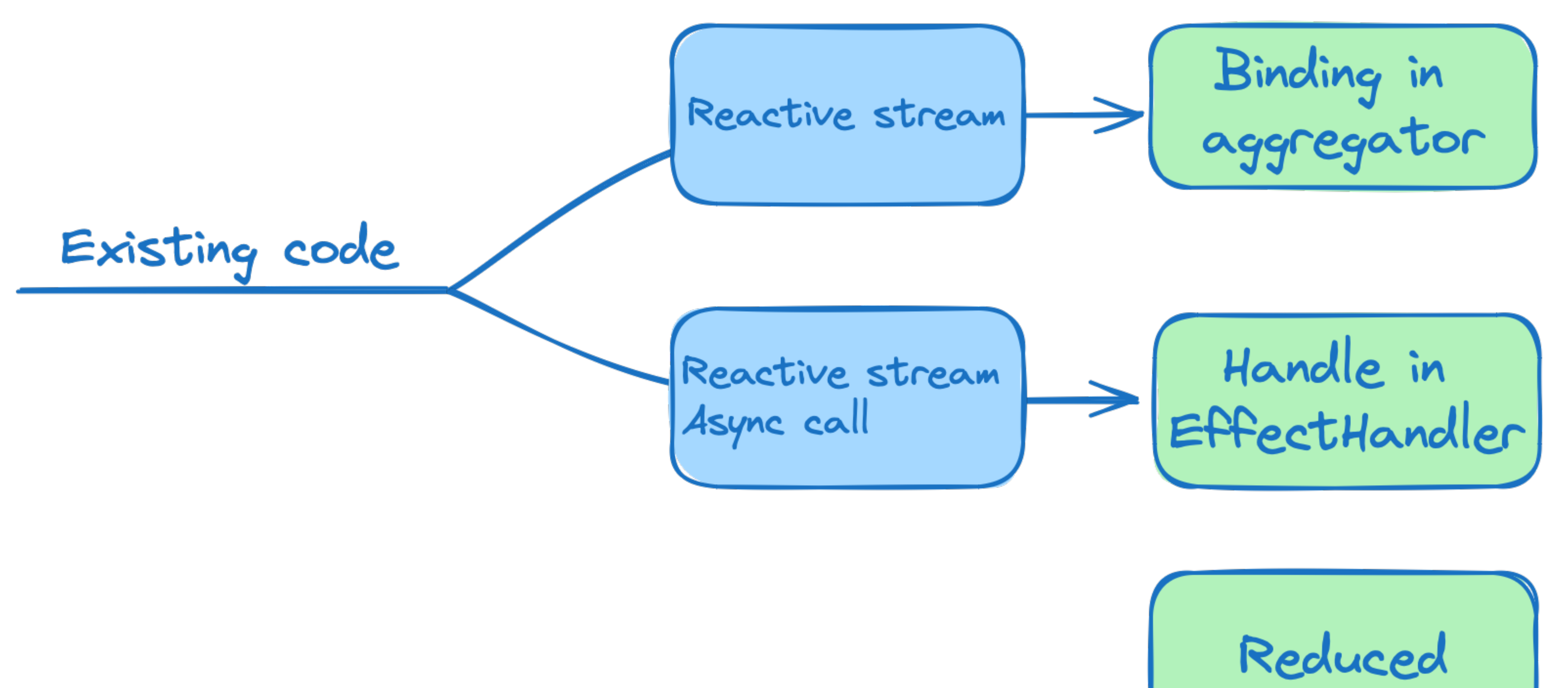
Integrate existed code

Pure fun

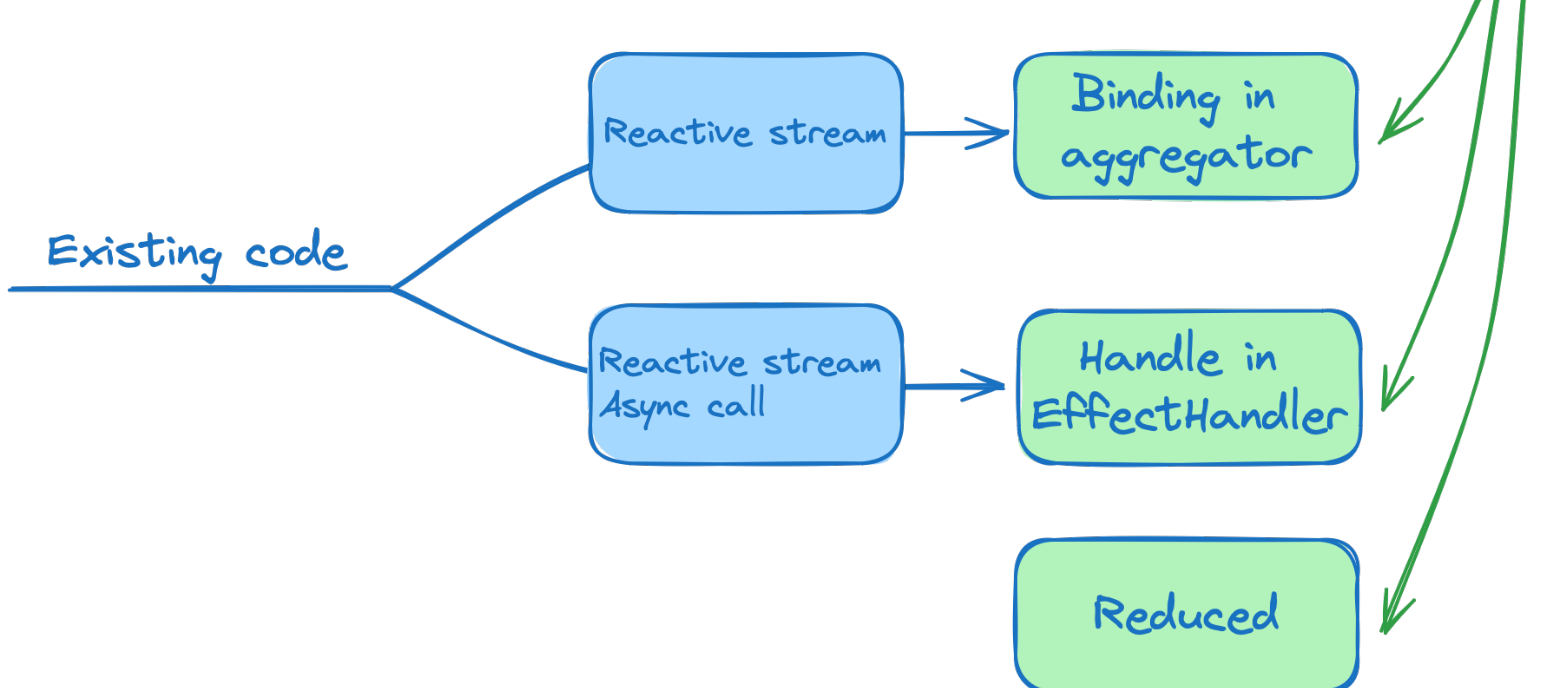


Integrate existed code

Pure fun



Integrate existed code



Migrate stateful component to Store

1. Find stateful component
2. Modulate a state
3. Pure fun → reducer
4. Other → EffectHandler
5. Build up a new store

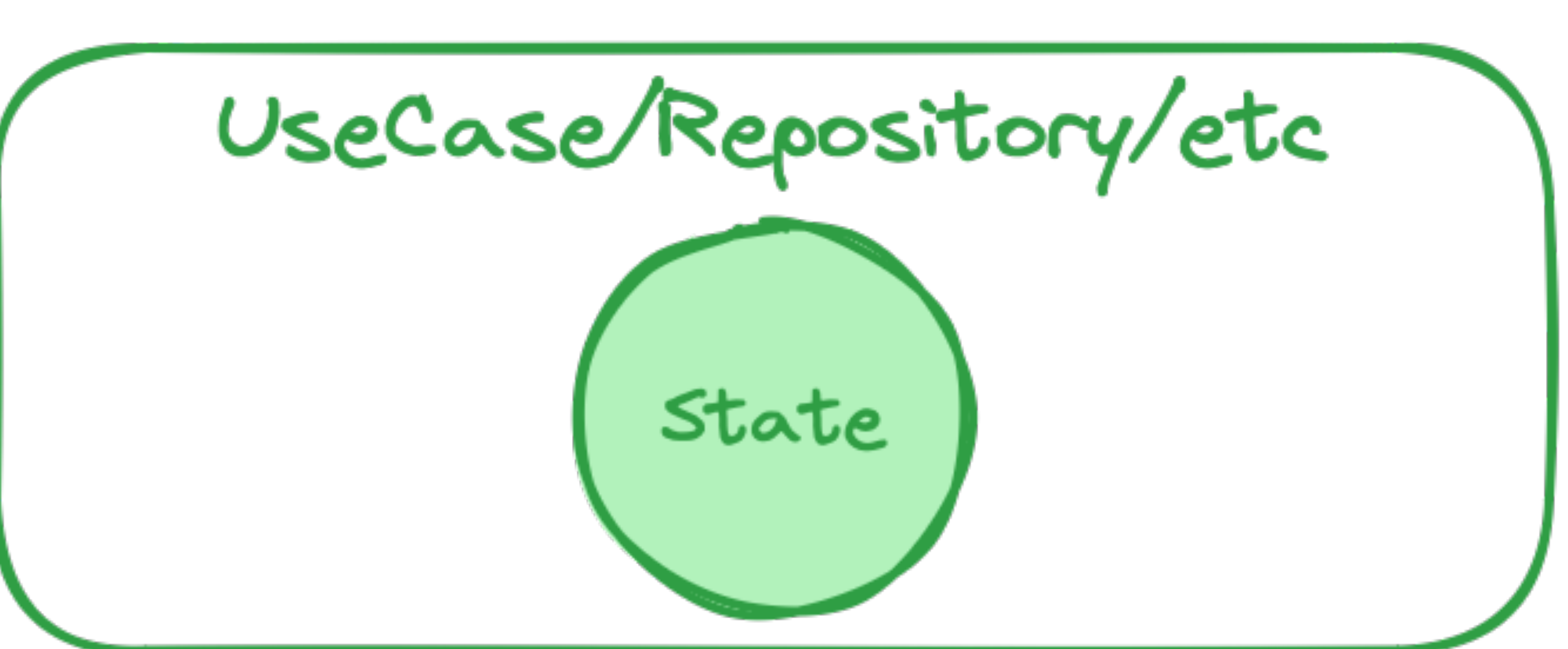
Migrate stateful component to Store

1. **Find stateful component**
2. Modulate a state
3. Pure fun → reducer
4. Other → EffectHandler
5. Build up a new store

UseCase/Repository/etc

Migrate stateful component to Store

1. Find stateful component
2. Modulate a state
3. Pure fun → reducer
4. Other → EffectHandler
5. Build up a new store

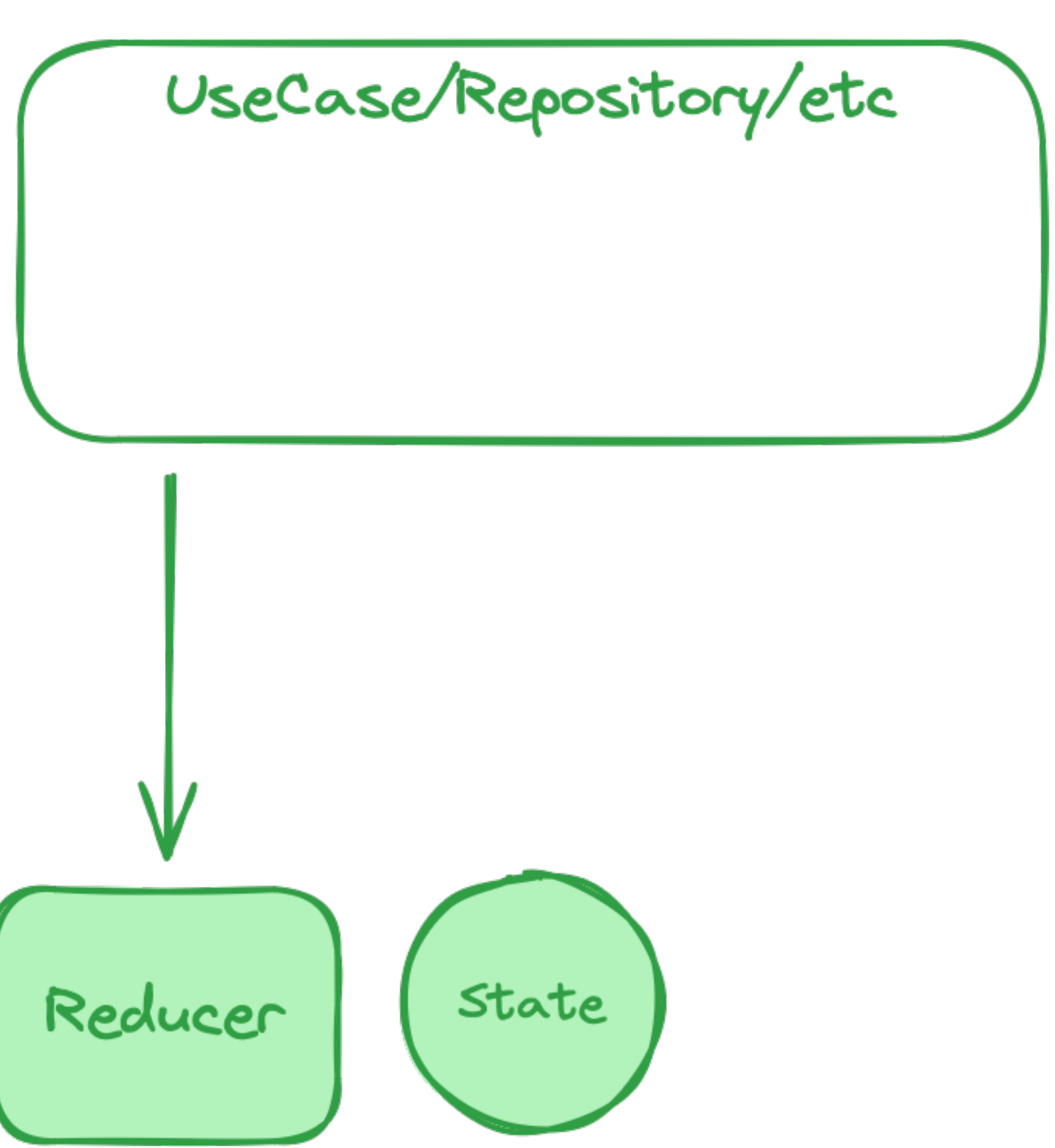


Migrate stateful component to Store

1. **Find stateful component**
2. **Modulate a state**
3. Pure fun → reducer
4. Other → EffectHandler
5. Build up a new store

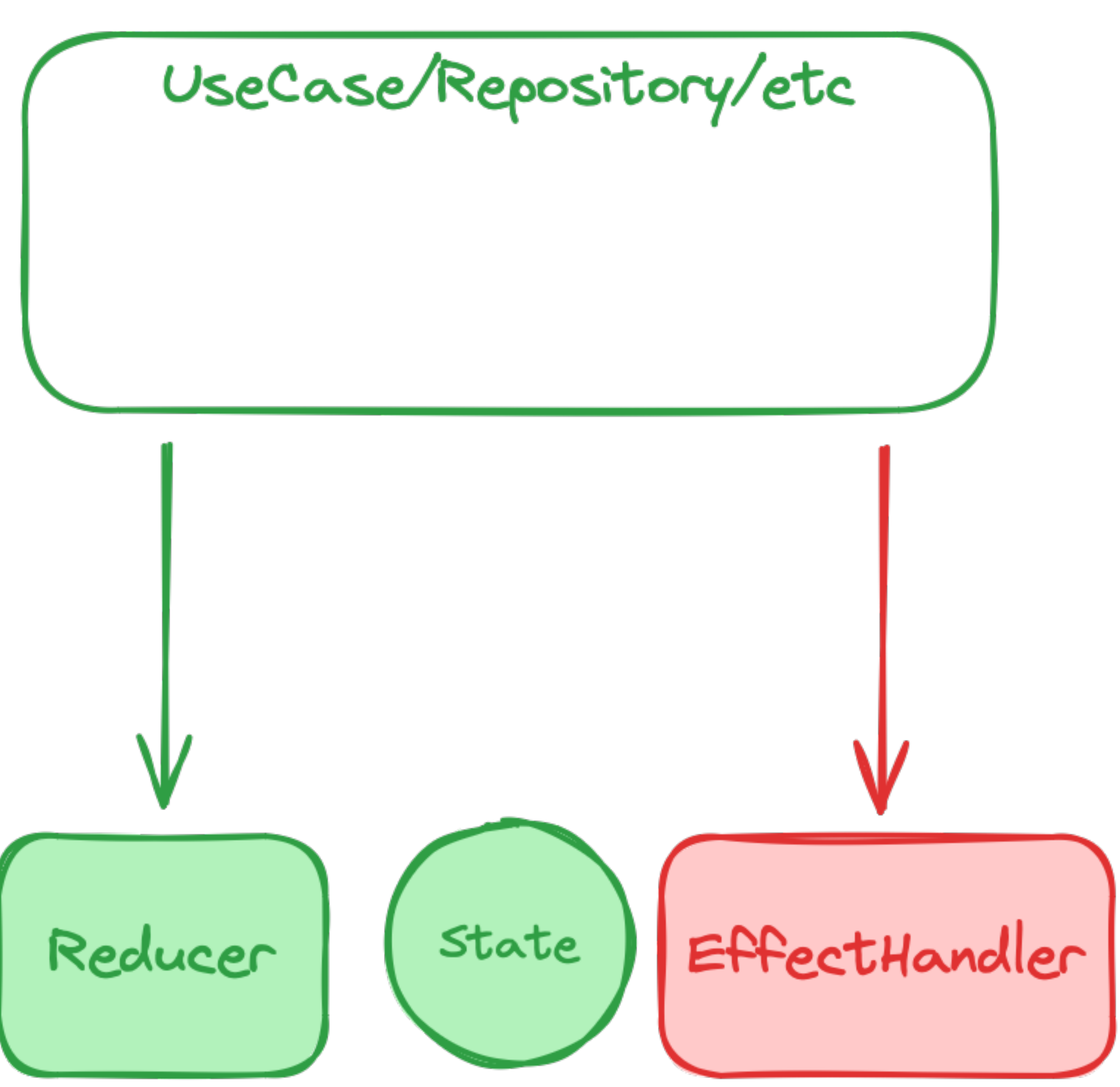
Migrate stateful component to Store

1. Find stateful component
2. Modulate a state
3. Pure fun → reducer
4. Other → EffectHandler
5. Build up a new store



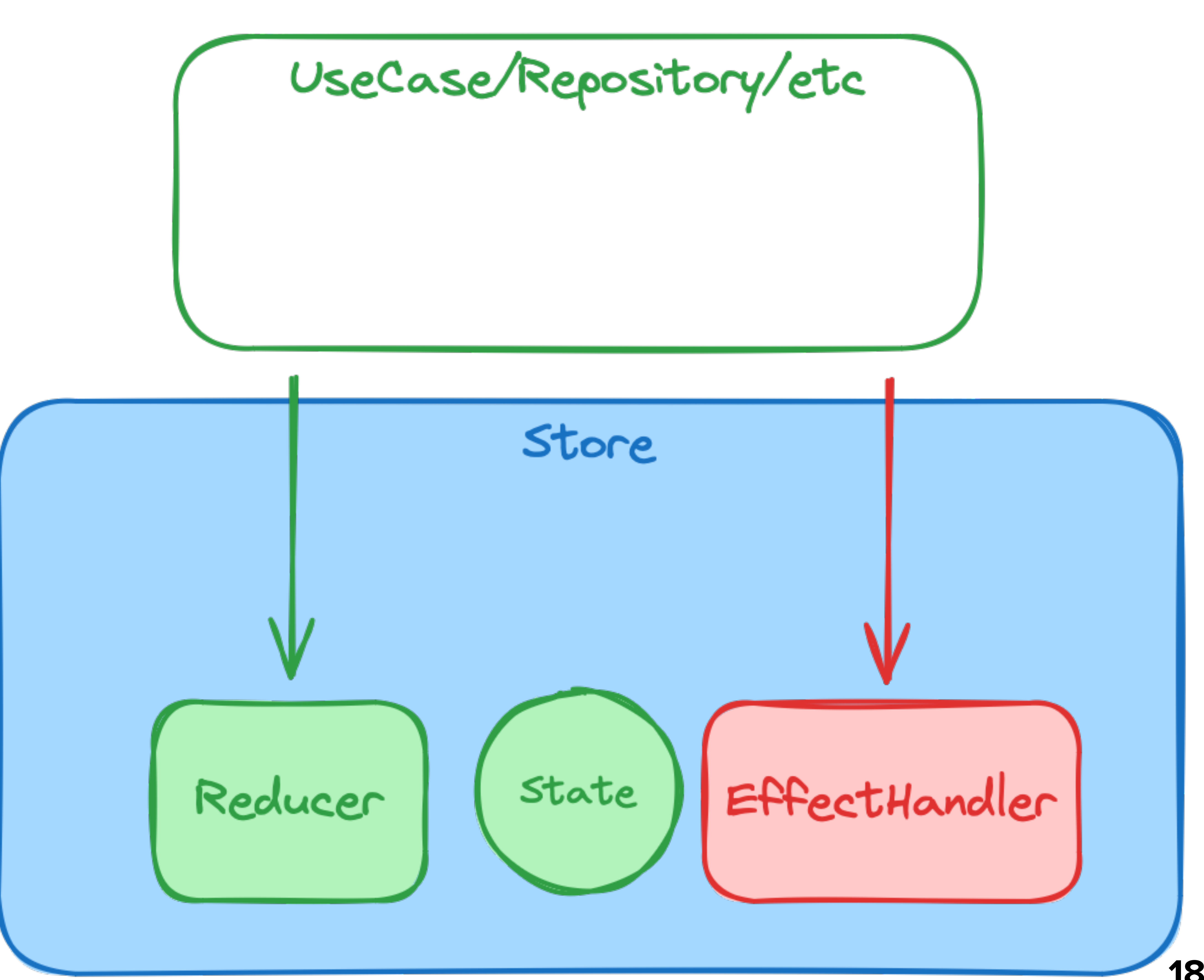
Migrate stateful component to Store

1. Find stateful component
2. Modulate a state
3. Pure fun → reducer
4. Other → EffectHandler
5. Build up a new store



Migrate stateful component to Store

1. Find stateful component
2. Modulate a state
3. Pure fun → reducer
4. Other → EffectHandler
5. Build up a new store



What we have learned

1. Handling long-running tasks
2. UI integration
3. Reusing effect handlers
4. Migration Tips & Tricks

Summary

1. Features granularity
2. How to use TEA in real life
3. Tone of samples
4. Tips & tricks
5. Ready to code architecture

Sample code

https://github.com/ikarenkov/Kombucha-UDF

Igor Karenkov

7 years in Android dev, TeamLead mobile-core @ HH

Develop open source (Modo & Kombucha-UDF)

Mentoring developers

Rock climber

@karenkovigor

ikarenkov

Mentoring - https://getmentor.dev/mentor/igor-karenkov-1058