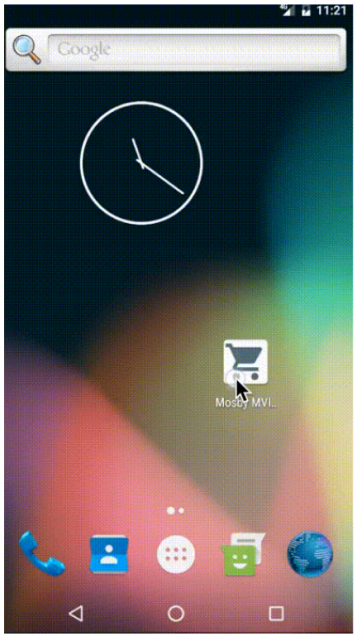
Reactive Apps with Model-View-Intent - Part 3: State Reducer


11:21
Google
Mosby MVi..

```java
public interface HomeView {

  /**
   * The intent to load the first page
   *
   * @return The value of the emitted item (boolean) can be ignored. true or false has no different meaning.
   */
  public Observable<Boolean> loadFirstPageIntent();

  /**
   * The intent to load the next page (pagination)
   *
   * @return The value of the emitted item (boolean) can be ignored. true or false has no different meaning.
   */
  public Observable<Boolean> loadNextPageIntent();

  /**
   * The intent to react on pull-to-refresh
   *
   * @return The value of the emitted item (boolean) can be ignored. true or false has no different meaning.
   */
  public Observable<Boolean> pullToRefreshIntent();

  /**
   * The intent to load more items from a given category
   *
   * @return Observable with the name of the category
   */
  public Observable<String> loadAllProductsFromCategoryIntent();

  /**
   * Renders the viewState
   */
  public void render(HomeViewState viewState);
}
```

The concrete View implementation is pretty straight forward and therefore I won't show the code here (can be found on github). Next let's focus on the Model. As already said in previous posts the Model should reflect the State. So let's introduce a Model called HomeViewState:

```
public final class HomeViewState {

  private final boolean loadingFirstPage; // Show the loading indicator instead of recyclerView
  private final Throwable firstPageError; // Show an error view if != null
  private final List<FeedItem> data;    // The items displayed in the recyclerview
  private final boolean loadingNextPage; // Shows the loading indicator for pagination
  private final Throwable nextPageError; // if != null, shows error toast that pagination failed
  private final boolean loadingPullToRefresh; // Shows the pull-to-refresh indicator
  private final Throwable pullToRefreshError; // if != null, shows error toast that pull-to-refresh failed

   // ... constructor ...
   // ... getters  ...
}
```

Note that FeedItem is just a interface every item has to implement that is displayable by the RecyclerView. For example Product implements FeedItem. Also the category title displayed in the recycler SectionHeader implements FeedItem. The UI element that indicates that more items of that category can be loaded is a FeedItem and holds internally it's own little State to indicate whether or not we are loading more Items of a certain category:

```
public class AdditionalItemsLoadable implements FeedItem {
  private final int moreItemsAvailableCount;
  private final String categoryName;
  private final boolean loading; // if true then loading items is in progress
  private final Throwable loadingError; // indicates an error has occurred while loading

   // ... constructor ...
   // ... getters  ...
```

And last but not least there is a business logic component HomeFeedLoader responsible to load FeedItems:

```
public class HomeFeedLoader {

  // Typically triggered by pull-to-refresh
  public Observable<List<FeedItem>> loadNewestPage() { ... }

  //Loads the first page
  public Observable<List<FeedItem>> loadFirstPage() { ... }

  // loads the next page (pagination)
  public Observable<List<FeedItem>> loadNextPage() { ... }

  // loads additional products of a certain category
  public Observable<List<Product>> loadProductsOfCategory(String categoryName) { ... }
}
```

So far so good, no big difference to how we have implemented the “search screen” as described in part 2. Now let’s try to add support for pull-to-refresh.

```
class HomePresenter extends MviBasePresenter<HomeView, HomeViewState> {

  private final HomeFeedLoader feedLoader;

  @Override protected void bindIntents() {
    //
    // In a real app some code here should rather be moved into an Interactor
    //
    Observable<HomeViewState> loadFirstPage = ... ;

    Observable<HomeViewState> pullToRefresh = intent(HomeView::pullToRefreshIntent)
        .flatMap(ignored -> feedLoader.loadNewestPage()
            .map( items -> new HomeViewState(...))
            .startWith(new HomeViewState(...))
            .onErrorReturn(error -> new HomeViewState(...)));

    Observable<HomeViewState> allIntents = Observable.merge(loadFirstPage, pullToRefresh);

    subscribeViewState(allIntents, HomeView::render);
  }
}
```

Use Observable.merge() to merge together multiple intents.

Now let's connect the dots step by step in our Presenter. Please note that some code shown here as part of the Presenter should rather be moved into an Interactor in a real world application (which I didn't for the sake of better readability). First, lets start with loading the initial data:

```
class HomePresenter extends MviBasePresenter<HomeView, HomeViewState> {

  private final HomeFeedLoader feedLoader;

  @Override protected void bindIntents() {
    //
    // In a real app some code here should rather be moved into an Interactor
    //
    Observable<HomeViewState> loadFirstPage = intent(HomeView::loadFirstPageIntent)
        .flatMap(ignored -> feedLoader.loadFirstPage()
            .map(items -> new HomeViewState(items, false, null) )
            .startWith(new HomeViewState(emptyList, true, null) )
            .onErrorReturn(error -> new HomeViewState(emptyList, false, error))

    subscribeViewState(loadFirstPage, HomeView::render);
  }
}
```

But wait: feedLoader.loadNewestPage() only returns “newer” items but what about the previous items we have already loaded? In “traditional” MVP one would call something like view.addNewItems(newItems) but we have already discussed in part 1 why this is a bad idea (“The State Problem”). The problem we are facing now is that pull-to-refresh depends on the previous HomeViewState since we want to “merge” the previous items with the items returned from pull-to-refresh.

State Reducer is a concept from functional programming that takes the previous state as input and computes a new state from the previous state like this:

```
public State reduce( State previous, Foo foo ){
  State newState;
  // ... compute the new State by taking previous state and foo into account ...
  return newState;
}
```

```
class HomePresenter extends MviBasePresenter<HomeView, HomeViewState> {

  private final HomeFeedLoader feedLoader;

  @Override protected void bindIntents() {
    //
    // In a real app some code here should rather be moved into an Interactor
    //
    Observable<PartialState> loadFirstPage = intent(HomeView::loadFirstPageIntent)
        .flatMap(ignored -> feedLoader.loadFirstPage()
            .map(items -> new PartialState.FirstPageData(items) )
            .startWith(new PartialState.FirstPageLoading(true) )
            .onErrorReturn(error -> new PartialState.FirstPageError(error))

    Observable<PartialState> pullToRefresh = intent(HomeView::pullToRefreshIntent)
        .flatMap(ignored -> feedLoader.loadNewestPage()
            .map( items -> new PartialState.PullToRefreshData(items)
            .startWith(new PartialState.PullToRefreshLoading(true)))
            .onErrorReturn(error -> new PartialState.PullToRefreshError(error)));

    Observable<PartialState> allIntents = Observable.merge(loadFirstPage, pullToRefresh);
    Observable<PartialState> allIntents = Observable.merge(loadFirstPage, pullToRefresh);
    HomeViewState initialState = ... ; // Show loading first page
    Observable<HomeViewState> stateObservable = allIntents.scan(initialState, this::viewStateReducer)

    subscribeViewState(stateObservable, HomeView::render);
  }

  private HomeViewState viewStateReducer(HomeViewState previousState, PartialState changes){
    ...
  }
}
```

So what we did here is, that each Intent now returns an Observable<PartialState> rather then directly Observable<HomeViewState>. Then we merge them all into one observable stream with Observable.merge() and finally apply the reducer function (Observable.scan()). Basically what this means is that, whenever the user starts an intent, this intent will produce PartialState objects which then will be "reduced" to a HomeViewState that then eventually will be displayed in the View (HomeView.render(HomeViewState)). The only missing part is the state reducer function itself. The HomeViewState class itself hasn't changed (scroll up to see the class definition), but we have added a Builder (Builder pattern) so that we can create new HomeViewState objects in a convenient way. So let's implement the state reducer function:

```
private HomeViewState viewStateReducer(HomeViewState previousState, PartialState changes){
    if (changes instanceof PartialState.FirstPageLoading)
        return previousState.toBuilder() // creates a new copy by taking the internal values of previousState
        .firstPageLoading(true) // show ProgressBar
        .firstPageError(null) // don't show error view
        .build()

    if (changes instanceof PartialState.FirstPageError)
     return previousState.builder()
         .firstPageLoading(false) // hide ProgressBar
         .firstPageError(((PartialState.FirstPageError) changes).getError()) // Show error view
         .build();

     if (changes instanceof PartialState.FirstPageLoaded)
       return previousState.builder()
           .firstPageLoading(false)
           .firstPageError(null)
           .data(((PartialState.FirstPageLoaded) changes).getData())
           .build();
    if (changes instanceof PartialState.PullToRefreshLoading)
     return previousState.builder()
           .pullToRefreshLoading(true) // Show pull to refresh indicator
           .nextPageError(null)
           .build();

   if (changes instanceof PartialState.PullToRefreshError)
     return previousState.builder()
         .pullToRefreshLoading(false) // Hide pull to refresh indicator
         .pullToRefreshError(((PartialState.PullToRefreshError) changes).getError())
         .build();

   if (changes instanceof PartialState.PullToRefreshData) {
     List<FeedItem> data = new ArrayList<>();
     data.addAll(((PullToRefreshData) changes).getData()); // insert new data on top of the list
     data.addAll(previousState.getData());
     return previousState.builder()
       .pullToRefreshLoading(false)
       .pullToRefreshError(null)
       .data(data)
       .build();
   }


  throw new IllegalStateException("Don't know how to reduce the partial state " + changes);
}
```

```
class HomePresenter extends MviBasePresenter<HomeView, HomeViewState> {

  private final HomeFeedLoader feedLoader;

  @Override protected void bindIntents() {

    //
    // In a real app some code here should rather be moved to an Interactor
    //

    Observable<PartialState> loadFirstPage = ... ;
    Observable<PartialState> pullToRefresh = ... ;

    Observable<PartialState> nextPage =
      intent(HomeView::loadNextPageIntent)
          .flatMap(ignored -> feedLoader.loadNextPage()
              .map(items -> new PartialState.NextPageLoaded(items))
              .startWith(new PartialState.NextPageLoading())
              .onErrorReturn(PartialState.NexPageLoadingError::new));

      Observable<PartialState> loadMoreFromCategory =
          intent(HomeView::loadAllProductsFromCategoryIntent)
              .flatMap(categoryName -> feedLoader.loadProductsOfCategory(categoryName)
                  .map( products -> new PartialState.ProductsOfCategoryLoaded(categoryName, products))
                  .startWith(new PartialState.ProductsOfCategoryLoading(categoryName))
                  .onErrorReturn(error -> new PartialState.ProductsOfCategoryError(categoryName, error)));
|
    Observable<PartialState> allIntents = Observable.merge(loadFirstPage, pullToRefresh, nextPage, loadMoreFromCategory);
    HomeViewState initialState = ... ; // Show loading first page
    Observable<HomeViewState> stateObservable = allIntents.scan(initialState, this::viewStateReducer)

    subscribeViewState(stateObservable, HomeView::render);
  }

  private HomeViewState viewStateReducer(HomeViewState previousState, PartialState changes){
    // ... PartialState handling for First Page and pull-to-refresh as shown in previous code snipped ...

      if (changes instanceof PartialState.NextPageLoading) {
       return previousState.builder().nextPageLoading(true).nextPageError(null).build();
      }

      if (changes instanceof PartialState.NexPageLoadingError)
        return previousState.builder()
            .nextPageLoading(false)
            .nextPageError(((PartialState.NexPageLoadingError) changes).getError())
            .build();


      if (changes instanceof PartialState.NextPageLoaded) {
        List<FeedItem> data = new ArrayList<>();
        data.addAll(previousState.getData());
         // Add new data add the end of the list
        data.addAll(((PartialState.NextPageLoaded) changes).getData());
        return previousState.builder().nextPageLoading(false).nextPageError(null).data(data).build();
      }

      if (changes instanceof PartialState.ProductsOfCategoryLoading) {
          int indexLoadMoreItem = findAdditionalItems(categoryName, previousState.getData());

          AdditionalItemsLoadable ail = (AdditionalItemsLoadable) previousState.getData().get(indexLoadMoreItem);

          AdditionalItemsLoadable itemsThatIndicatesError = ail.builder() // creates a copy of the ail item
          .loading(true).error(null).build();

          List<FeedItem> data = new ArrayList<>();
          data.addAll(previousState.getData());
          data.set(indexLoadMoreItem, itemsThatIndicatesError); // Will display a loading indicator

          return previousState.builder().data(data).build();
       }

      if (changes instanceof PartialState.ProductsOfCategoryLoadingError) {
        int indexLoadMoreItem = findAdditionalItems(categoryName, previousState.getData());

        AdditionalItemsLoadable ail = (AdditionalItemsLoadable) previousState.getData().get(indexLoadMoreItem);

        AdditionalItemsLoadable itemsThatIndicatesError = ail.builder().loading(false).error( ((ProductsOfCategoryLoadingError)changes)
'
        List<FeedItem> data = new ArrayList<>();
        data.addAll(previousState.getData());
        data.set(indexLoadMoreItem, itemsThatIndicatesError); // Will display an error / retry button

        return previousState.builder().data(data).build();
      }

      if (changes instanceof PartialState.ProductsOfCategoryLoaded) {
        String categoryName = (ProductsOfCategoryLoaded) changes.getCategoryName();
        int indexLoadMoreItem = findAdditionalItems(categoryName, previousState.getData());
        int indexOfSectionHeader = findSectionHeader(categoryName, previousState.getData());

        List<FeedItem> data = new ArrayList<>();
        data.addAll(previousState.getData());
        removeItems(data, indexOfSectionHeader, indexLoadMoreItem); // Removes all items of the given category

        // Adds all items of the category (includes the items previously removed)
        data.addAll(indexOfSectionHeader + 1,((ProductsOfCategoryLoaded) changes).getData());

        return previousState.builder().data(data).build();
      }

      throw new IllegalStateException("Don't know how to reduce the partial state " + changes);
  }
}
```