Reactive Apps with Model-View-Intent - Part 2: View and Intent

```
class PersonsPresenter extends Presenter<PersonsView> {

  public void load(){
    getView().showLoading(true); // Displays a ProgressBar on the screen

    backend.loadPersons(new Callback(){
      public void onSuccess(List<Person> persons){
        getView().showPersons(persons); // Displays a list of Persons on the screen
      }

      public void onError(Throwable error){
        getView().showError(error); // Displays a error message on the screen
      }
    });
  }
}
```

we should create a “Model” that reflects the “State”:

```
class PersonsModel {
  // In a real application fields would be private
  // and we would have getters to access them
  final boolean loading;
  final List<Person> persons;
  final Throwable error;

  public(boolean loading, List<Person> persons, Throwable error){
    this.loading = loading;
    this.persons = persons;
    this.error = error;
  }
}
```

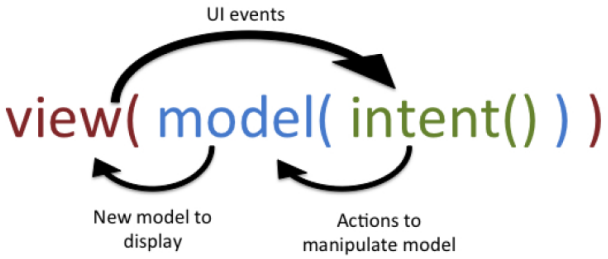
And then the Presenter could be implemented like this:

```
class PersonsPresenter extends Presenter<PersonsView> {

  public void load(){
    getView().render( new PersonsModel(true, null, null) ); // Displays a ProgressBar

    backend.loadPersons(new Callback(){
      public void onSuccess(List<Person> persons){
        getView().render( new PersonsModel(false, persons, null) ); // Displays a list of Persons
      }

      public void onError(Throwable error){
          getView().render( new PersonsModel(false, null, error) ); // Displays a error message
      }
    });
  }
}
```

Now the View has a Model which then will be “rendered” on the screen simply by invoking render(personsModel). In the first part we also talked about the importance of an unidirectional data flow and that your business logic should drive this model. Before we start to connect the dots lets quickly discuss the main idea of MVI.

Model-View-Intent (MVI)

This pattern was specified by André Medeiros (Staltz) for a JavaScript framework he has written called cycle.js. From a theoretical (and mathematical) point of view we could describe Model-View-Intent as follows:



- intent(): This function takes the input from the user (i.e. UI events, like click events) and translate it to "something" that will be passed as parameter to model() function. This could be a simple string to set a value of the model to or more complex data structure like an Object. We could say we have the intention to change the model with an intent.
- model(): The model() function takes the output from intent() as input to manipulate the Model. The output of this function is a new Model (state changed). So it should not update an already existing Model.
- view(): This method takes the model returned from model() function and gives it as input to the view() function. Then the View simply displays this Model somehow. view() is basically the same as view.render(model).

Connecting the dots with RxJava

We want that our data flows unidirectional. Here comes RxJava into play. Do we need RxJava to build reactive apps with an unidirectional data flow or MVI based apps? No, we could also write imperative and procedural code. However, RxJava is very good for event based programming. Since UI are event based too using RxJava makes a lot of sense.

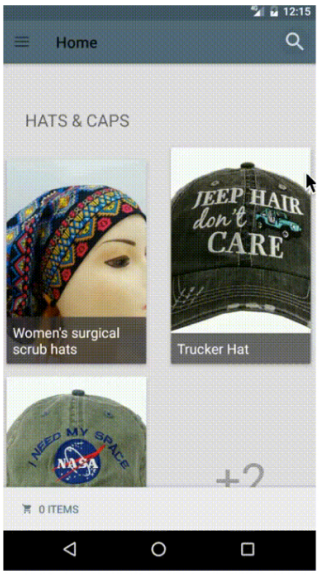
```
public interface SearchViewState {

  /**
   * The search has not been stared yet
   */
  final class SearchNotStartedYet implements SearchViewState {
  }

  /**
   * Loading: Currently waiting for search result
   */
  final class Loading implements SearchViewState {
  }

  /**
   * Indicates that the search has delivered an empty result set
   */
  final class EmptyResult implements SearchViewState {
    private final String searchQueryText;

    public EmptyResult(String searchQueryText) {
      this.searchQueryText = searchQueryText;
    }

    public String getSearchQueryText() {
      return searchQueryText;
    }
  }

  /**
   * A valid search result. Contains a list of items that have matched the searching criteria.
   */
  final class SearchResult implements SearchViewState {
    private final String searchQueryText;
    private final List<Product> result;

    public SearchResult(String searchQueryText, List<Product> result) {
      this.searchQueryText = searchQueryText;
      this.result = result;
    }

    public String getSearchQueryText() {
      return searchQueryText;
    }

    public List<Product> getResult() {
      return result;
    }
  }

  /**
   * Indicates that an error has occurred while searching
   */
  final class Error implements SearchViewState {
    private final String searchQueryText;
    private final Throwable error;

    public Error(String searchQueryText, Throwable error) {
      this.searchQueryText = searchQueryText;
      this.error = error;
    }

    public String getSearchQueryText() {
      return searchQueryText;
    }

    public Throwable getError() {
      return error;
    }
  }
}
```

```
class SearchViewState {
  Throwable error; // if not null, an error has occurred
  boolean loading; // if true loading data is in progress
  List<Product> result; // if not null this is the result of the search
  boolean SearchNotStartedYet; // if true, we have the search not started yet
}
```

```
public class SearchInteractor {
  final SearchEngine searchEngine; // Makes http calls

  public Observable<SearchViewState> search(String searchString) {
    // Empty String, so no search
    if (searchString.isEmpty()) {
      return Observable.just(new SearchViewState.SearchNotStartedYet());
    }

    // search for products
    return searchEngine.searchFor(searchString) // Observable<List<Product>>
        .map(products -> {
          if (products.isEmpty()) {
            return new SearchViewState.EmptyResult(searchString);
          } else {
            return new SearchViewState.SearchResult(searchString, products);
          }
        })
        .startWith(new SearchViewState.Loading())
        .onErrorReturn(error -> new SearchViewState.Error(searchString, error));
  }
}
```

```java
public interface SearchView {

  /**
   * The search intent
   *
   * @return An observable emitting the search query text
   */
  Observable<String> searchIntent();

  /**
   * Renders the View
   *
   * @param viewState The current viewState state that should be displayed
   */
  void render(SearchViewState viewState);
}
```

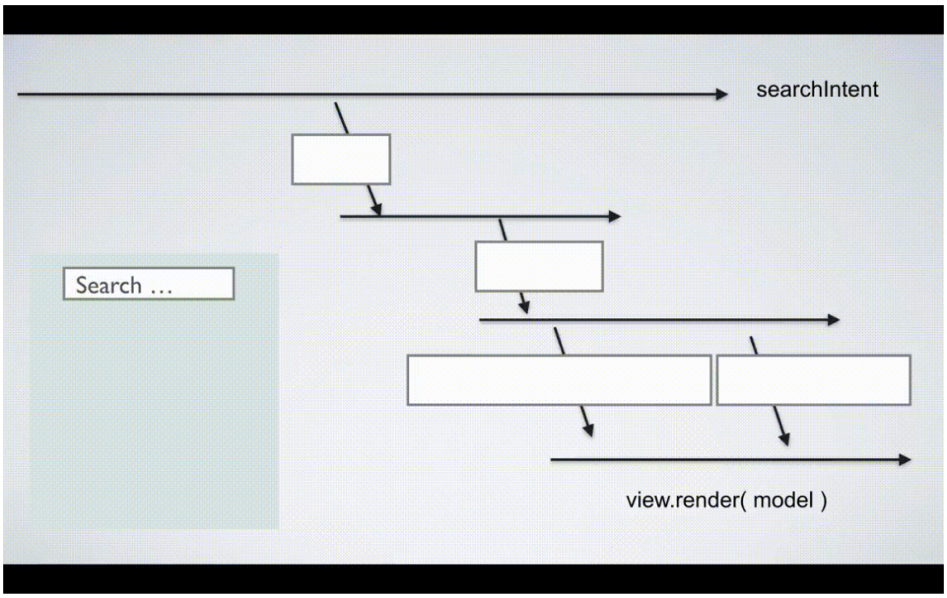
In this case our View only offers one intent but in general a View could offer multiple intents. In part 1 we have discussed why a single render() function is a nice approach, if it is unclear why we should prefer a single render() you should read part 1 again (or leave a comment below; see also comment section in part 1). Before we start with the concrete implementation of the View layer, let's take a look how the final result should look like:



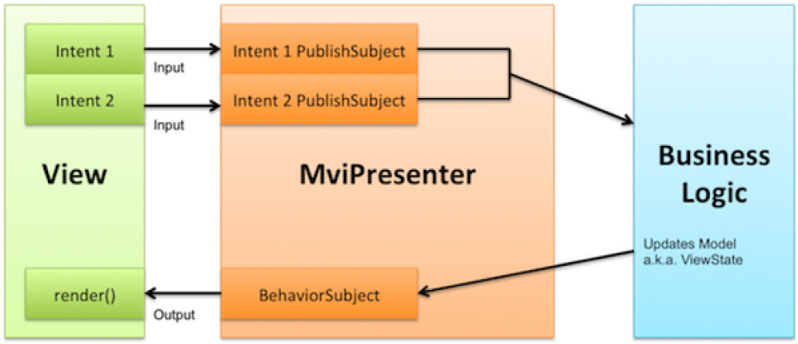
```
public class SearchFragment extends Fragment implements SearchView {

  @BindView(R.id.searchView) android.widget.SearchView searchView;
  @BindView(R.id.container) ViewGroup container;
  @BindView(R.id.loadingView) View loadingView;
  @BindView(R.id.errorView) TextView errorView;
  @BindView(R.id.recyclerView) RecyclerView recyclerView;
  @BindView(R.id.emptyView) View emptyView;
  private SearchAdapter adapter;

  @Override public Observable<String> searchIntent() {
    return RxSearchView.queryTextChanges(searchView) // Thanks Jake Wharton :)
        .filter(queryString -> queryString.length() > 3 || queryString.length() == 0)
        .debounce(500, TimeUnit.MILLISECONDS);
  }

  @Override public void render(SearchViewState viewState) {
    if (viewState instanceof SearchViewState.SearchNotStartedYet) {
      renderSearchNotStarted();
    } else if (viewState instanceof SearchViewState.Loading) {
      renderLoading();
    } else if (viewState instanceof SearchViewState.SearchResult) {
      renderResult(((SearchViewState.SearchResult) viewState).getResult());
    } else if (viewState instanceof SearchViewState.Error) {
      renderError();
    } else {
      throw new IllegalArgumentException("Don't know how to render viewState " + viewState);
    }
  }

  private void renderResult(List<Product> result) {
    TransitionManager.beginDelayedTransition(container);
    recyclerView.setVisibility(View.VISIBLE);
    loadingView.setVisibility(View.GONE);
    emptyView.setVisibility(View.GONE);
    errorView.setVisibility(View.GONE);
    adapter.setProducts(result);
    adapter.notifyDataSetChanged();
  }

  private void renderSearchNotStarted() {
    TransitionManager.beginDelayedTransition(container);
    recyclerView.setVisibility(View.GONE);
    loadingView.setVisibility(View.GONE);
    errorView.setVisibility(View.GONE);
    emptyView.setVisibility(View.GONE);
  }

  private void renderLoading() {
    TransitionManager.beginDelayedTransition(container);
    recyclerView.setVisibility(View.GONE);
    loadingView.setVisibility(View.VISIBLE);
    errorView.setVisibility(View.GONE);
    emptyView.setVisibility(View.GONE);
  }

  private void renderError() {
    TransitionManager.beginDelayedTransition(container);
    recyclerView.setVisibility(View.GONE);
    loadingView.setVisibility(View.GONE);
    errorView.setVisibility(View.VISIBLE);
    emptyView.setVisibility(View.GONE);
  }

  private void renderEmptyResult() {
    TransitionManager.beginDelayedTransition(container);
    recyclerView.setVisibility(View.GONE);
    loadingView.setVisibility(View.GONE);
    errorView.setVisibility(View.GONE);
    emptyView.setVisibility(View.VISIBLE);
  }
}
```

```
public class SearchPresenter extends MviBasePresenter<SearchView, SearchViewState> {
  private final SearchInteractor searchInteractor;

  @Override protected void bindIntents() {
    Observable<SearchViewState> search =
        intent(SearchView::searchIntent)
            .switchMap(searchInteractor::search) // I have used flatMap() in the video above, but switchMap() makes more sense here
            .observeOn(AndroidSchedulers.mainThread());

    subscribeViewState(search, SearchView::render);
  }
}
```

Please node that SearchView::searchIntent is just a Java 8 shorthand for searchView.searchIntent()


searchIntent
Search …
view.render(model)

The rule is simple: use intent() to “wrap” any intent of the view. Use subscribeViewState() instead of Observable.subscribe(…).



The counter part to bindIntent() is unbindIntents() which is invoked exactly one time the View is destroyed permanently. For instance putting a fragment on the back stack doesn’t destroy the View permanently, but finishing an Activity does. Since intent() and subscribeViewState() already take care of subscription management you only barely need to implement unbindIntents().

What about other lifecycle events like onPause() or onResume()? I still think that Presenters don’t need lifecycle Events. However, if you really think you need them you can simply see a lifecycle event like onPause() as an intent. Your View could offer a pauseIntent() which is triggered by android lifecycle instead of a user interaction intent like clicking on a button. But both are valid intents.