Implementation details and guidance

`SDK related code`

- Don't have any dependency to third party library.
- Using Swift struct, enum, protocol oriented programming, closure, generics.
- Go with what's offer by swift and apple native frameworks.
- Maintain reusability and portability

`Redux`

> The easiest way to think about redux is, that in app you can only have one single copy of the `same` data share by multiple view controller that wants latest update of given piece data. Redux is for avoid duplicate data state in multiple view controller, and issues by having multiple copy of the same data in each view controllers, and data can going out of sync, if screen flow and data logic involve is complicated and tend to change or expand over time. There should be only one implementation for same data processing/transformation logic in `reducer`.

- `State` changes/update would be push out to any view controller or view that subscriber/listening to data changes. It follows `pub/sub` pattern.
- `State` is the single point of truth in app for any specific data that need to be share by multiple controllers, like for example user input and selection. And data is not view related.
- `Reducer` implementation need to be generic and **NOT** include any customer specific business logic.
- `State`, `Action` need to be define as **struct**, class type can't use for `state`, `action`, otherwise it would cause memory related issue with redux.
- No property is allowed in `reducer`. Reducers group into struct in app is purely for code structure purpose and confine them to their respective scope. By not presenting all `reducer` functions as global swift function, are to prevent code collision in global scope, and make Xcode autocomplete easier to find any giving reducer function.
- `Reducer` should not contain any hard coded value inside functions, adding it would introducer hidden state and create inconsistency.
- Implementation of `reducer` need to be purely functional with no side effect. Which mean reducer function always take some parameters as input and return updated result as output. There are no object oriented programming in reducers. In `reducer` implementation swift work the same way as other functional programing languages.
- `Protocol` can be use with `action`, `reducer` where it make sense, to enforce common requirement for actions and to simplify reducer data processing logic.
- `Reducer` logics always running in `main thread`, no async operation is allowed. As async operation can break the order of state changes and create inconsistency in state. `State` change and update need to follow `FIFO` (First In First Out).

- Heavy data processing, async operation like API call need to happen `outside of reducer`.. Either as view controller extension, or as common data class or services. `Reducer` should only do lightweight data processing and data transformation. As long running operation would block redux data flow and main thread.
- `Reducer` are close loop that process/transform incoming data and changes from `Action`, and result is return back to state. Inject any closure from outside into reducer dynamically to alter functions behaviors is undesirable.

UI

- RxSwift, RxCocoa, Rx... should use mostly with UI where it make sense, like validation user input and UI interaction...

  > **Note:** Of course Rx can be use for data related tasks too and use everywhere for everything. But data tasks in most case would work just fine without Rx, if it can be done with what swift and apple frameworks have to offer. But if data operation is fairly complex, where Rx do offer advantage over native swift/apple approach, and produce cleaner and more maintainable, reliable code, then Rx approach is prefer. Decision need to take case by case for what's best in giving context.

- General rule for UI is `clear separation of code`, make them reusable and maintainable as possible, strip data logic in view controller out to SDK, Redux, Extension, Utility / Help function, Service components.
- Split views in view controller into view components/segments where it's possible, like `Forms`.. In `Interface Builder`, avoid single view controller contain too many `subviews` as nested view objects.
- Don't mix UI code with Data related code as much as possible, data logic inside view controller is hard to reuse and debug, and write unit/functional test around them.
- In `View Controller` / `View` contain most/only UI related code, make it dumb, reusable and flexible as possible. Extract action / bindings / glue code to extension.
- Shouldn’t contain more then around 300 lines of code in any files, most of them would be far less the 300 lines. Break down codes into file groups, use what is most logical as separation point.
- With `redux` view controller don’t move/pass data from one controller to another controller. Instead view controller just subscribe any data change from `redux state`.
- Push to another view controller should use router action. Work the same way as with web link, view controller can go from where they are to anywhere, which mean no `segues`.

Q&A

1. Why Redux(ReSwift) ?

   In many occasion same piece of data being share and use in multiple screens, redux is the only viable way to keep data consistent across multiple screens over time. As user can view, change, update and modify data from multiple place in app. With reducer there would be single

implementation how data being process. Data logic is testable and implementation is reusable for similar projects.

2. All data should be stored in state or there is some data that can be stored in controllers ?

   All data should be stored in state. Redux `subscriber` in view controller is a function. When the function call end, then latest state values that push to view controller is gone. In view controller if you want a copy of any giving data for later use than don't in sync with redux update schedule. Then you can put a copy of data inside view controller, sometimes it's needed for UI validation and enforce business rule in UI layer.. But any changes to data should stored in state as single point of truth.

3. All logic regarding data processing should be located in the reduces and not in the controllers?

   All logic regarding data processing and transformation should located in reducers, but for SDK/API call, UI logics and business rule validation need to stay outside in controllers, mostly as VC extensions or as Utility / Help function and Service components.

4. We don't inject any data directly into controller, controller should populate all needed data from states, right?

   Not always, it depend on context.. If it's data related to user input and selection, or data need to be shareable/up-to-date cross multiple screens, then it should store in redux states.

   View/UI related data that is tie to individual screen can stay inside each view controller, like color, view position, button state.. or sensor related data like GPS position update don't need to stored in states either.