Mobile Computing

Flutter
App Architectures

App Architecture

. There are some organization difficulties in managing State and Business Logic (Domain Logic, linked to data, and Application Logic, linked to functionalities/user operations) in Flutter applications.

. It is not easy to adapt well known architectures, like MVC, to the Flutter app structure.

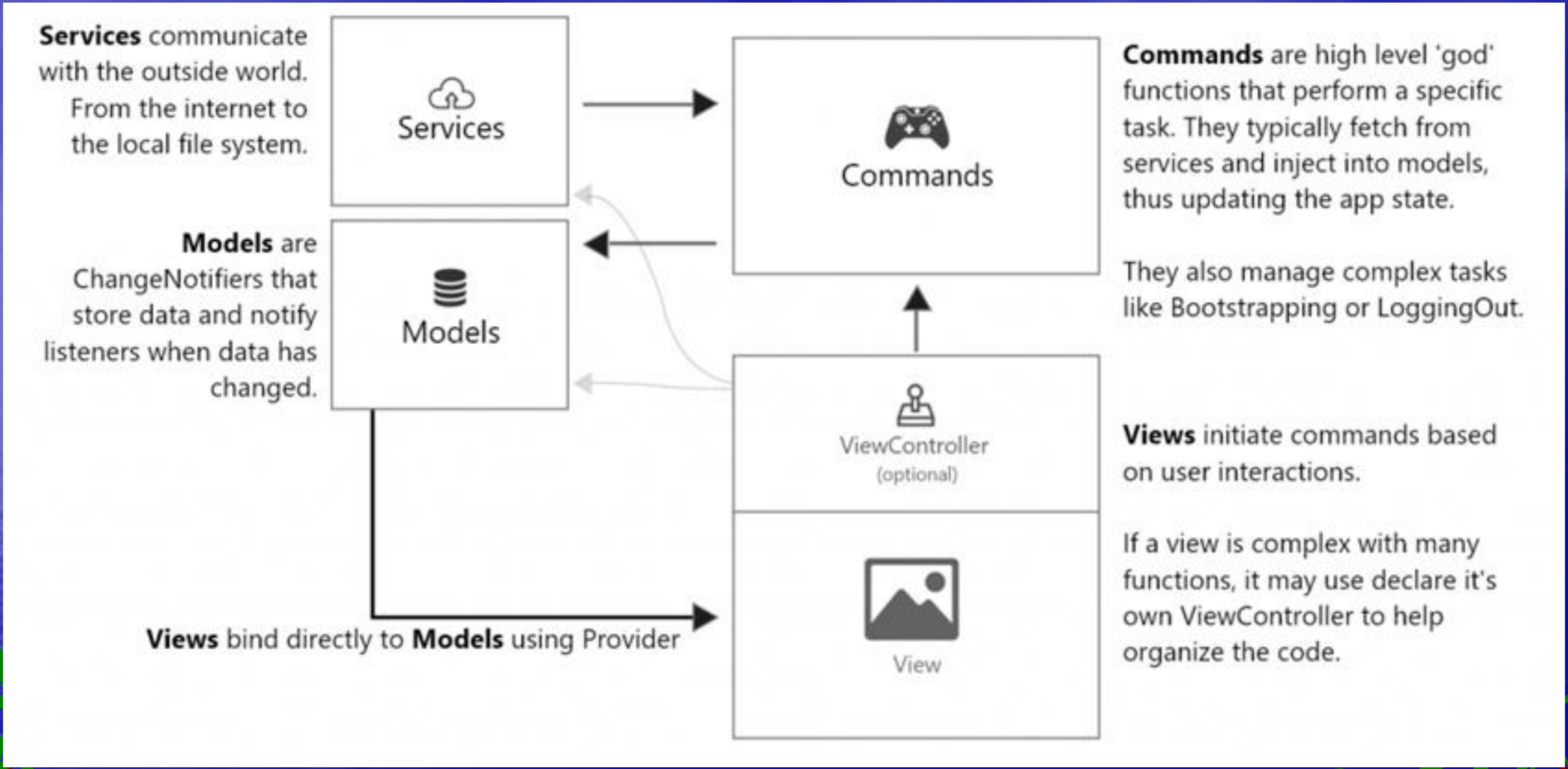
. Many architectural patterns have been proposed with corresponding plugins … (https://flutter.dev/docs/development/data-and-backend/state-mgmt/options)

. One of them leverages the Flutter class ChangeNotifier (notifies if data changes) and the package Provider (binds Models to the BuildContext) to implement a clean MVC+S architecture.
(https://blog.gskinner.com/archives/2020/09/flutter-state-management-with-mvcs.html)

. The MVC+S architecture includes the:
- . Models – Holds the application state and simple operations to access/filter/manipulate that data. These operations are also called the Domain Logic.
- . Views – the UI and pages build with the Flutter widgets. This layer may also contain the "view controllers" with handlers for local interactions.
- . Controller – contains the Application Logic composed by the business operations that constitute the functionalities of the application. These operations can be organized in Commands invoked by the other layers, but mainly is result of UI interactions.
- . Services – contain the operations in the outside world. Usually, they are invoked by Commands that can retrieve data and inject it on Models, or in the other direction.

MVC+S Architecture

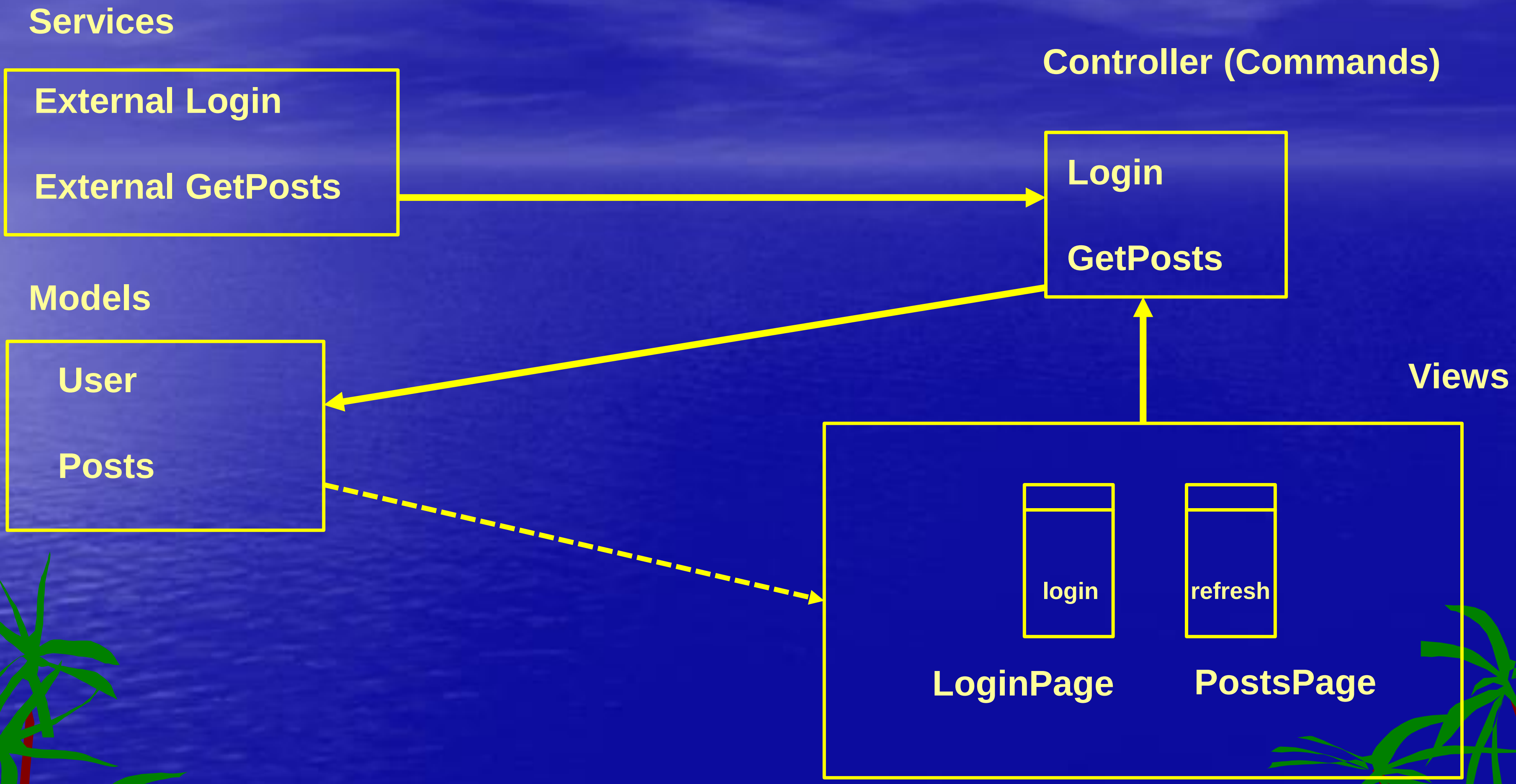
MVC+S Diagram


Services communicate with the outside world. From the internet to the local file system.
Services
Commands
Commands are high level 'god' functions that perform a specific task. They typically fetch from services and inject into models, thus updating the app state.
They also manage complex tasks like Bootstrapping or LoggingOut.
Models are ChangeNotifiers that store data and notify listeners when data has changed.
Models
ViewController (optional)
Views initiate commands based on user interactions.
If a view is complex with many functions, it may use declare it's own ViewController to help organize the code.
View
Views bind directly to **Models** using Provider

The implementation of this architecture requires access to the Models in every place in the App. For doing that, a special widget is provided in an external package. It associates Models and Services with the application context.

Provider package: (https://pub.dev/packages/provider)

Example


Services
External Login
External GetPosts
Controller (Commands)
Login
GetPosts
Models
User
Posts
Views
login
refresh
LoginPage
PostsPage

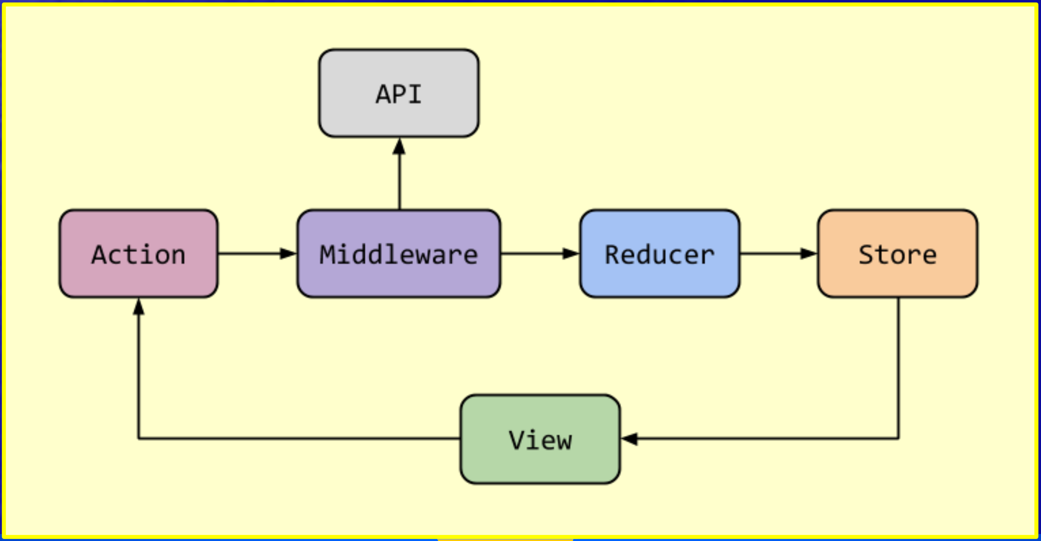
Redux Architecture

The Redux architecture specifies that all the application state is centralized in the Store block (it can contain several Models)

The View (flutter widget tree) generates Actions from user interactions, that specify external requests and/or state modifications. The external requests are filtered out by a Middleware block. The other Actions go to a Reducer function.

The Reducer function modifies the Store, according to the specified Action.

Views have access to the Store, and whenever this is modified it triggers a View redrawing, taking into account the app state inside the Store.

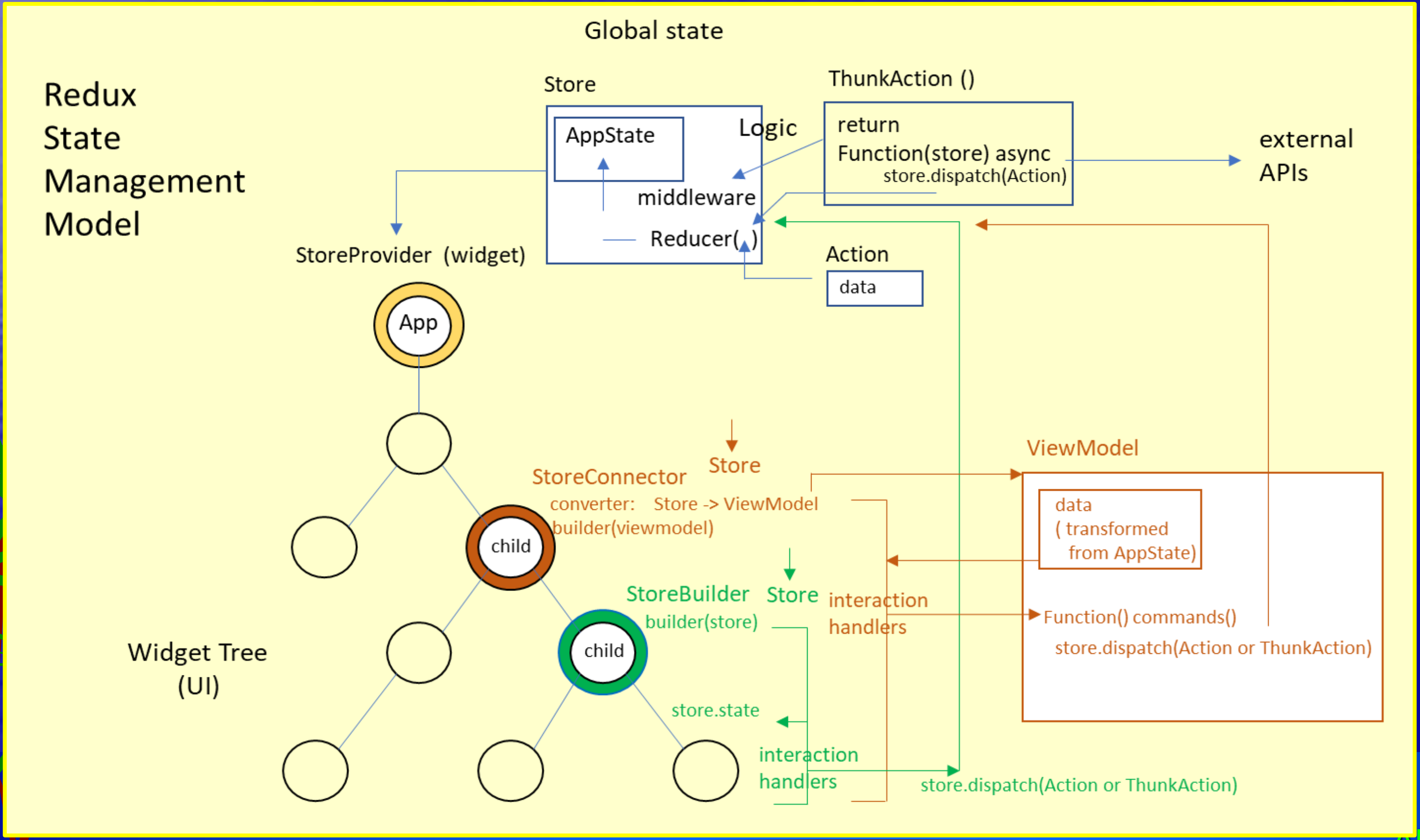


Redux packages

The Redux architecture needs some external packages to be implemented in Flutter.

The base functionality comes from the redux package which implements the Store class.
Redux package: (https://pub.dev/packages/redux)

The middleware layer is implemented by the redux_thunk package.
And some widgets to implement access to the Store and trigger the redrawing of Views are implemented in the flutter_redux package.


Global state
Redux State Management Model
Store
AppState
Logic
middleware
Reducer( )
ThunkAction ()
return Function(store) async
store.dispatch(Action)
external APIs
Action
data
StoreProvider (widget)
App
child
child
StoreConnector
Store
converter: Store -> ViewModel
builder(viewmodel)
ViewModel
data (transformed from AppState)
Function() commands()
store.dispatch(Action or ThunkAction)
StoreBuilder
Store
builder(store)
interaction handlers
store.state
interaction handlers
store.dispatch(Action or ThunkAction)
Widget Tree (UI)