**React:**

* Front End library developed by facebook
* Used for handling view layer(UI) of web and mobile apps
* Component based architecture
* Implements **one-way reactive data flow**
* Flux or Redux help to keep the unidirection data flow
* **Performance** - Used Virtual DOM
* **Declarative** in nature
* React also used **ES6 syntax**

**JSX -** Javascript syntax extension

React uses JSX for templating instead of regular JavaScript. It is not necessary to use it, but there are some pros that comes with it.

* JSX is faster because it performs optimization while compiling code to JavaScript.
* It is also type-safe and most of the errors can be caught during compilation.
* JSX makes it easier and faster to write templates if you are familiar with HTML.

import React from 'react';

class App extends React.Component {

render() {

return (

<div>Hello World!!!</div>

);

}

} export default App;

* Nested Elements - *<div>…..</div>*
* Custom Attributes – *data-myattribute*
* JS expression – *{ }*

**Styling**

* React components used inline-styling

import React from 'react';

class App extends React.Component {

render() {

var myStyle = {

fontSize: 100,

color: '#FF0000'

}

return (

<div><h1 style = {myStyle}>Header</h1></div>

);

}

} export default App;

**Components:**

* **Stateless example**
* **State**
* **Props**
* **Component life-cycle**
* **React forms**
* **Comunication**

**Props and State**

Two types of data that control a component: **props** and **state**.

Props are set by the parent and they are fixed throughout the lifetime of a component. For data that is going to change, we have to use State.

**State**: minimal representation of data in app.

* Every thing that changes in app including UI State or Data contain in single object **State**. Hence all state of app is in a single JS object.
* All state change or explicit i.e all changes are trackable

**Flux and Redux**

**Store :**

Hold the current state

Only way to change the state is to emit/dispatch an [action](http://redux.js.org/docs/Glossary.html#action), an object describing what happened.

As actions are just plain objects, they can be logged, serialized, stored, and later replayed for debugging or testing purposes

**Dispatcher:**

This is central hub of the app. All the data is dispatched and sent to the stores.

**Actions**: minimal representation of change in data. Actions are simply JavaScript objects that use **type** property to inform about the data that should be sent to the store.

* State tree is read only, to change state component dispatch actions – action is JS object represent change of data
* Must have **type** property
* Action can be user interaction or or any network request.

**Pure & Impure functions:**

* **Pure**: return value depend on arguments
  + No DB or network calls
  + They are predictable
  + They donot modified their arguments
* **Impure**: Just opposite to pure functions.

**Reducers: (**to describe state mutation we need reducers**)(**manage the state update**)**

* Reducers are pure function that take previous state and action and returns the next state of the applicaton.

*Function function\_name(state, action){*

*If(typeof state ===’undefined’){*

*Return intialState;*

*}*

*if(action.type){*

*return newState;*

*}else{*

*return state;*

*}*

*}*

* It doesnot modify the state it return the new state as it is pure function
* It is not slow to return new state object as it store refrence to unchanged

*CreateStore* function of redux accept reducer function as argument

*getState* return the current value of state

*dispatch* is used to dispatch an action

*susbscribe(a callback)* called any time an action is dispatched

**Flux:**

Flux is a design pattern for handling data(state) in your application.

**Problem:** View update model, model update view and model can update other model these interdependencies result in cascading changes and difficult to reproduce bugs.

**Solution:** Unidirectional data flow

Image

**Action Creator**

*Think of the action creator as a* ***telegraph operator****. You go to the action creator knowing basically what message you want to send, and then the action creator formats that in a way that the rest of the system can understand.*

Image

* Charge of formating actions with *type(MESSAGE\_CREATE / MESSAGE\_READ)* and *payload(has all info that require to perform the action)*
* Passed the created action to the dispatcher

 A new developer can come on the project, open up the action creator files and see the entire API - all of the possible state changes — that your system provides.

**Dispatcher**

*Think of the dispatcher like* ***a telephone switchboardoperator*** *at a phone switchboard. It keeps a list of all of the stores that it needs to send actions to. When an action comes in from the action creator, it will pass the action around to different stores.*

Image

* A big registry of callbacks
* Send action to all registered stores via callback(in sequence)

**Store**

*Think of the store as an* ***over-controlling bureaucrat****. All state changes must be made by it personally. And you can’t directly request that it change the state. There are no setters on the store. To request a state change, you must follow proper procedure… you must submit an action via the action creator/dispatcher pipeline.*

Image

* Hold the state of the application
* Also hold the all state change logic
* Registered with dispatcher to accept actions
* No setters, only way in is through actions
* Check the action type to whether it cares about this one or not, and changes the state accordingly.
* Once any change occur emits the state change event(allow the subscribe *Controller views* to know the state change)

**Controller View and Sub Views**

*Think of  view is a* ***presenter****. It isn’t aware of anything in the application, it just knows the data that’s handed to it and how to format the data into output*

*Think of controller view is like a* ***middle manager*** *between the store and the view. The store tells it when the state has changed. It collects the new state and then passes the updated state along to all of the views under it.*

Image

* Take the change state from store(controller view) pass to all child views so that they can re-render itself
* View accept actions/inputs

**Flux Isuues**

1. **Hot Reloading** – **Wipe out data**

In Flux, the store contains two things:

1. State
2. State change logic.

So when we reload the store object to see the effect that the new state change logic has, you lose the state that the store is holding on to. Plus, you mess up the event subscriptions that tie the store to the rest of the system.

**Solution**  
Separate these two functions. Have one object that holds on to the state. This object doesn’t get reloaded. Have another object that contains all of the state change logic.

1. **State Rewritten**

We need to keep track of the states for travel debugging but as JS works it creates the same pointer to the previous old state whenever we change the state or make a copy of it.

**Solution**  
When an action comes in to the store, don’t handle it by changing the state. Instead, copy the state and make changes to the copy(immutable.js).

1. **Logging**

Let’s say you want to log every action as it comes in, and then log the state that results from it. In flux we need to to subscribe to the dispatcher’s updates to get updates from each store.

Image

**Solution**

Wrap part of system in other objects that add their bit of own functionality on top.

Use a tree to structure the state change logic. This makes it so the store only emits one event to notify the views that the state has changed.

Image

1. **Use of third party plugins**

No good place to just drop the plugin, extra custom efforts reuired to fit a plugin

**Redux**

* Redux reduces boilerplate and it makes it easier to reuse logic in the store.
* Unlike Flux, action creators in Redux do not send the action to the dispatcher. Instead, they return a formatted action object.
* In Redux, there is only one store
* **Reducers:**
* **Redux** have smart and dumb components instead of what we called controller view and view in flux. Smart components are in charge of the actions. If a dumb component underneath them needs to trigger an action, the smart component passes a function in via the props. The dumb component can then just treat that as a callback.
* React-redux – A view layer binding to connect the store to the views
  + Provider component
  + Connect()
  + Selector
* **Root component**

**React Native change folder/project name:**

* First copy the directory which your to-be-name-changed application exists. And go to your newly cloned directory.
* Change the name at index.ios/android.js file which is given as a parameter to AppRegistry.
* Change the name and version accordingly on package.json
* delete /ios and /android folders which are remaining from your older app.
* run $react-native upgrade to generate /ios and /android folders again.
* run $react-native link for any external dependency.
* finally run $react-native run-ios or anything you want.
* **Note**: also need to delte the older files in ios project

**Raeact native modules**

React Native documentation defines a *native module* as “an Objective-C class that implements the RCTBridgeModule protocol.” (RCT is an abbreviation for ReaCT.)

**Basic “Hello, World” module** - Chapter 7 Orielly learning React native

An Objective-C module must do the following in order to be available in React Native:

* Import the RCTBridgeModule header
* Declare that your module implements the RCTBridgeModule interface
* Call the RCT\_EXPORT\_MODULE() macro
* Have at least one method that is exported using the RCT\_EXPORT\_METHOD macro

**Redux- thunk**

**Redux middleware** to call actions asynchronously

Redux Thunk teaches Redux to recognize special kinds of actions that are in fact functions:

**Redux-persist**

it’s a wrapper/library for a store that automatically saves a redux state to the storage.