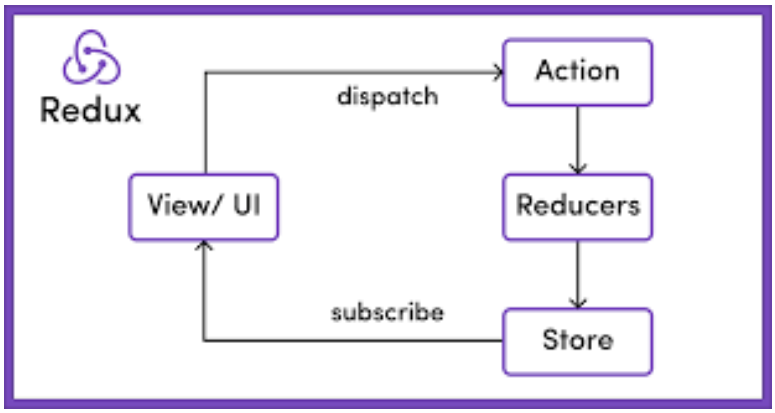
State Management

▼ Redux



dispatch(action): Dispatches an action. This is the only way to trigger a state change.
The store's reducer function will be called with the current `getState()` result and the given `action` synchronously. Its return value will be considered the next state

An **action** is a plain JavaScript object that has a `type` field. **You can think of an action as an event that describes something that happened in the application**. They may also contain additional data (payload) that is necessary to perform that action. Actions describe what happened but do not describe how the application's state changes.

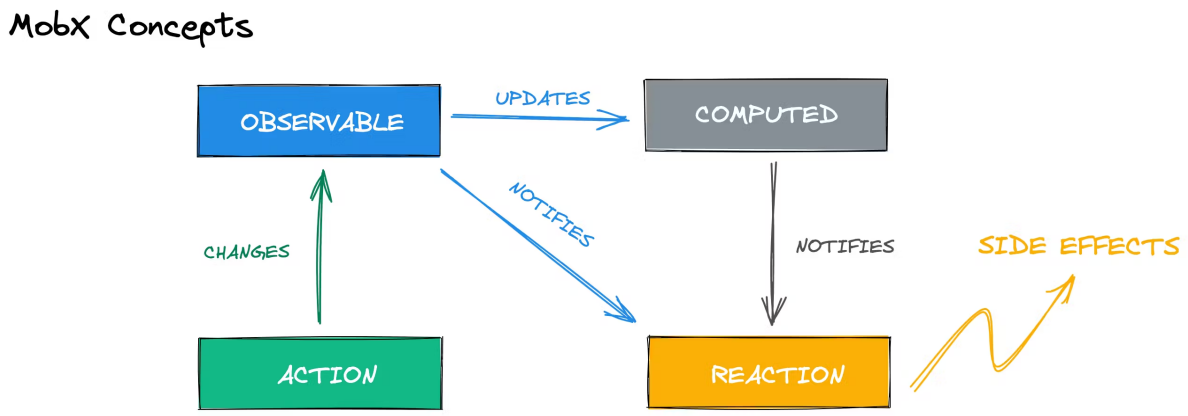
Reducers are functions that take the current `state` and an `action` as arguments, and return a new `state` result. In other words, `(state, action) => newState`. They are used to update the state of an application in response to an action.

A **store** holds the whole state tree of your application. The only way to change the state inside it is to dispatch an action on it, which triggers the root reducer function to calculate the new state.

A store is not a class. It's just an object with a few methods on it.

subscribe: Adds a change listener. It will be called any time an action is dispatched, and some part of the state tree may potentially have changed. You may then call `getState()` to read the current state tree inside the callback.

▼ Mobx



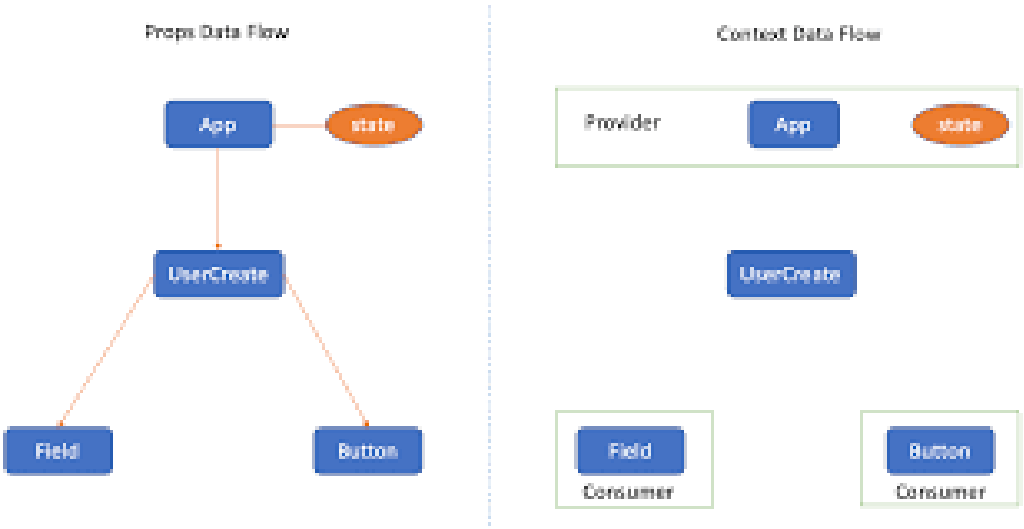
The `observable` annotation can also be called as a function to make an entire object observable at once. Observables are the state containers in MobX. They represent the application state that can change over time.

All applications have `actions`. An action is any piece of code that modifies the state. In principle, actions always happen in response to an event. For example, a button was clicked, some input changed, a websocket message arrived, etc.

`Computed` values can be used to derive information from other observables. They update automatically when the underlying observables they depend on change. Computed values are cached and recalculated only when necessary.

`Reactions` are functions that perform side effects in response to state changes. They are used for triggering functions that depends on the state. Reactions run every time the observables they depend on change.

▼ Context API (React)

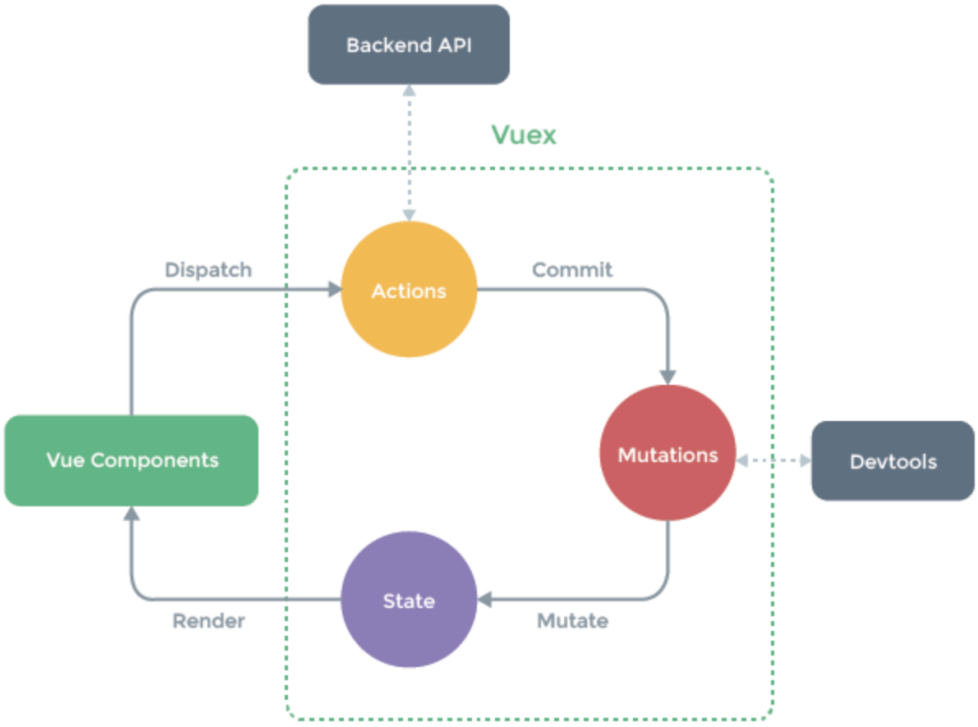


Usually, you will pass information from a parent component to a child component via props. But passing props can become verbose and inconvenient if you have to pass them through many components in the middle, or if many components in your app need the same information.

Context lets the parent component make some information available to any component in the tree below it—no matter how deep—without passing it explicitly through props.

Context lets a parent component provide data to the entire tree below it.

▼ Vuex



Vuex is a **state management pattern + library** for Vue.js applications. It serves as a centralized store for all the components in an application, with rules ensuring that the state can only be mutated in a predictable fashion.

Vuex uses a **single state tree** - that is, this single object contains all your application level state and serves as the "single source of truth."

It is similar to Redux.

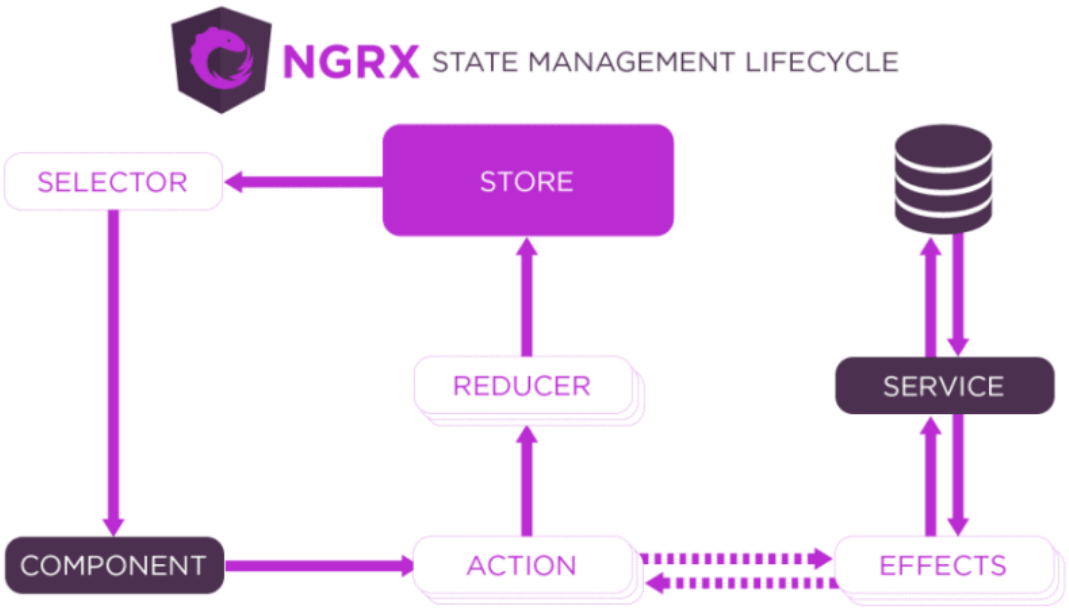
Actions are similar to mutations, the differences being that:

- Instead of mutating the state, actions commit mutations.
- Actions can contain arbitrary asynchronous operations.

Actions are triggered with the `store.dispatch` method

Mutations: The only way to actually change state in a Vuex store is by committing a mutation. Vuex mutations are very similar to events: each mutation has a string **type** and a **handler**.

▼ NgRx



Store is RxJS powered global state management for Angular applications, inspired by Redux. Store is a controlled state container designed to help write performant, consistent applications on top of Angular.

- Actions describe unique events that are dispatched from components and services.
- State changes are handled by pure functions called reducers that take the current state and the latest action to compute a new state.
- Selectors are pure functions used to select, derive and compose pieces of state. Used to update the component
- Effects are where you handle tasks such as fetching data, long-running tasks that produce multiple events, and other external interactions where your components don't need explicit knowledge of these interactions.

▼ Zustand

We use the create method for creating the Zustand store. We utilise the set function given as a parameter. The set method is used to access state and update the state. We define the state `count` and the increment function `inc`.

We can simply use the `inc` function and `store` state in the component with the help of the defined `useStore` hook.

```
import { create } from 'zustand'

// Define the useStore hook by creating a zustand store
const useStore = create((set) => ({
  count: 1,
  inc: () => set((state) => ({ count: state.count + 1 })),
}))

function Counter() {
    // Use the useStore hook to access count and inc from the
  const { count, inc } = useStore()
  return (
    <div>
      <span>{count}</span>
      <button onClick={inc}>one up</button>
    </div>
  )
}
```