# Rudex Reducer Rules

1. They should only calculate the new state value based on the state and action arguments.
2. They are not allowed to modify the existing state. Instead, they must copy the existing state and make changes to the copied values.
3. They must not do any asynchronous logic or have other “side effects”.

By asynchronous logic or “side effects”, we mean anything that the function does aside from returning a value, e.g., logging to the console, saving a file, setting a timer, making an HTTP request, and generating random numbers.

# Immutable Updates and Pure Functions

In programming, the three rules of reducers in Redux can be described more broadly. These rules state that reducers must perform **immutable** updates and be **pure functions**.

When a function makes immutable updates to its arguments, it doesn’t directly modify the original argument. Instead, it creates a copy and modifies that copy. This process is referred to as *immutable* updating because the function doesn’t alter or *mutate* the original arguments.

# Store

So far we have covered state, actions, reducers, and how they participate in the one-way data flow. Where does all of this take place?

Redux uses a special object called the **store**. The store serves as a container for the state, and it is the centerpiece of your application and the single source of truth. The store is in charge of facilitating the dispatching of actions, and triggering the reducer when actions are dispatched. In most Redux applications, there is typically only one store.

Let’s rephrase the data flow using the new term:

1. The store initializes the state with a default value.
2. The view displays that state to the user.
3. When a user interacts with the view, such as clicking a button, an action is dispatched to the store.
4. The store’s reducer combines the dispatched action and the current state to determine the next state.
5. The view is updated to display the new state.

Remember, Redux applications are built upon a one-way flow of data model and are managed by the store:

* The state is the set of data values that describes the application. It is used to render the user interface (UI).
* Users interact with the UI, which dispatches actions to the store. An action is an object that expresses a desired change to the state.
* The store generates its next state using a reducer function, which receives the most recent action and the current state as inputs.
* Finally, the UI is re-rendered based on the new state of the store, and the entire process can begin again.

# Revision

* Redux is a library for managing and updating application state based on the Flux architecture
* Redux makes code more predictable, testable, and maintainable by consolidating state in a single object. Components are just given data to render and can request changes using events called actions.
* In a Redux application, data flows in one direction: from state to view to action back to state, and so on.
* State is the current information behind a web application.
* An action is an object describing an event in the application. It must have a type property and it typically has a payload property as well.
* A reducer is a function that determines the application’s next state given a current state and a specific action. It returns a default initial state if none is provided and returns the current state if the action is not recognized
* A reducer must follow these three rules:
  1. They should only calculate the new state value based on the existing state and action.
  2. They are not allowed to modify the existing state. Instead, they must copy the existing state and make changes to the copied values.
  3. They must not do any asynchronous logic or other “side effects”.
* In other words, a reducer must be a pure function, and it must update the state immutably.
* The store is a container for state, it provides a way to dispatch actions, and it calls the reducer when actions are dispatched. Typically, there is only one store in a Redux application.

### Connect the Redux Store to a UI

Connecting a Redux store with any UI requires a few consistent steps, regardless of how the UI is implemented:

* Create a Redux store
* Render the initial state of the application.
* Subscribe to updates. Inside the subscription callback:
  + Get the current store state
  + Select the data needed by this piece of UI
  + Update the UI with the data
* Respond to UI events by dispatching Redux actions