ELECTRON STATE MANAGEMENT LIBRARY

SMOL-STORE

MAIN PROCESS AND RENDERER PROCESS

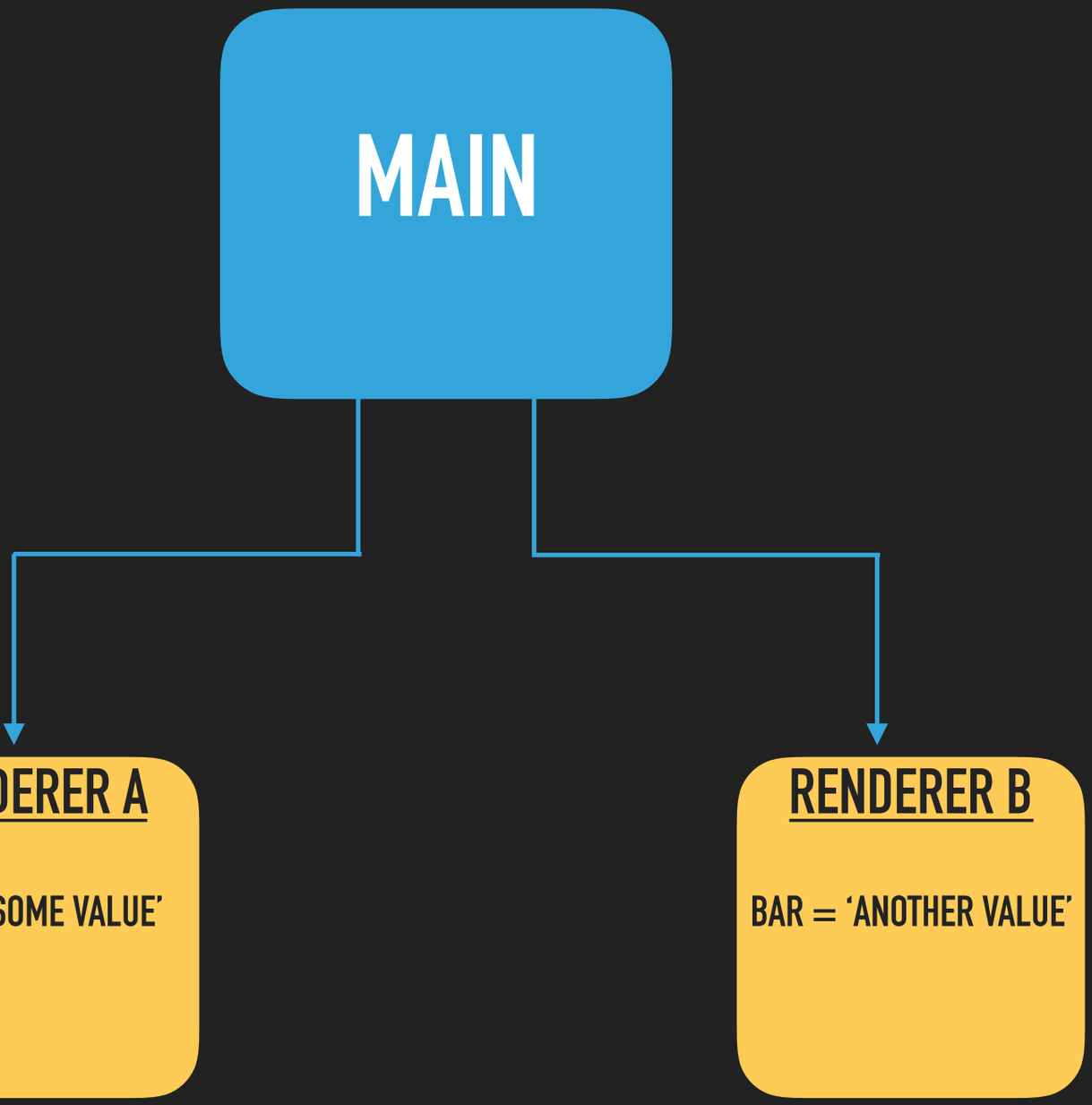
- The main process is the control center and heart of the app. As long as the main process is running, your app is as well. There can only be one main process
- Renderer processes run on a separate thread from the main, and are responsible for rendering the app's UI. Multiple renderer processes can be running at a given time
- The main process spawns renderer processes

COMMUNICATION BETWEEN PROCESSES

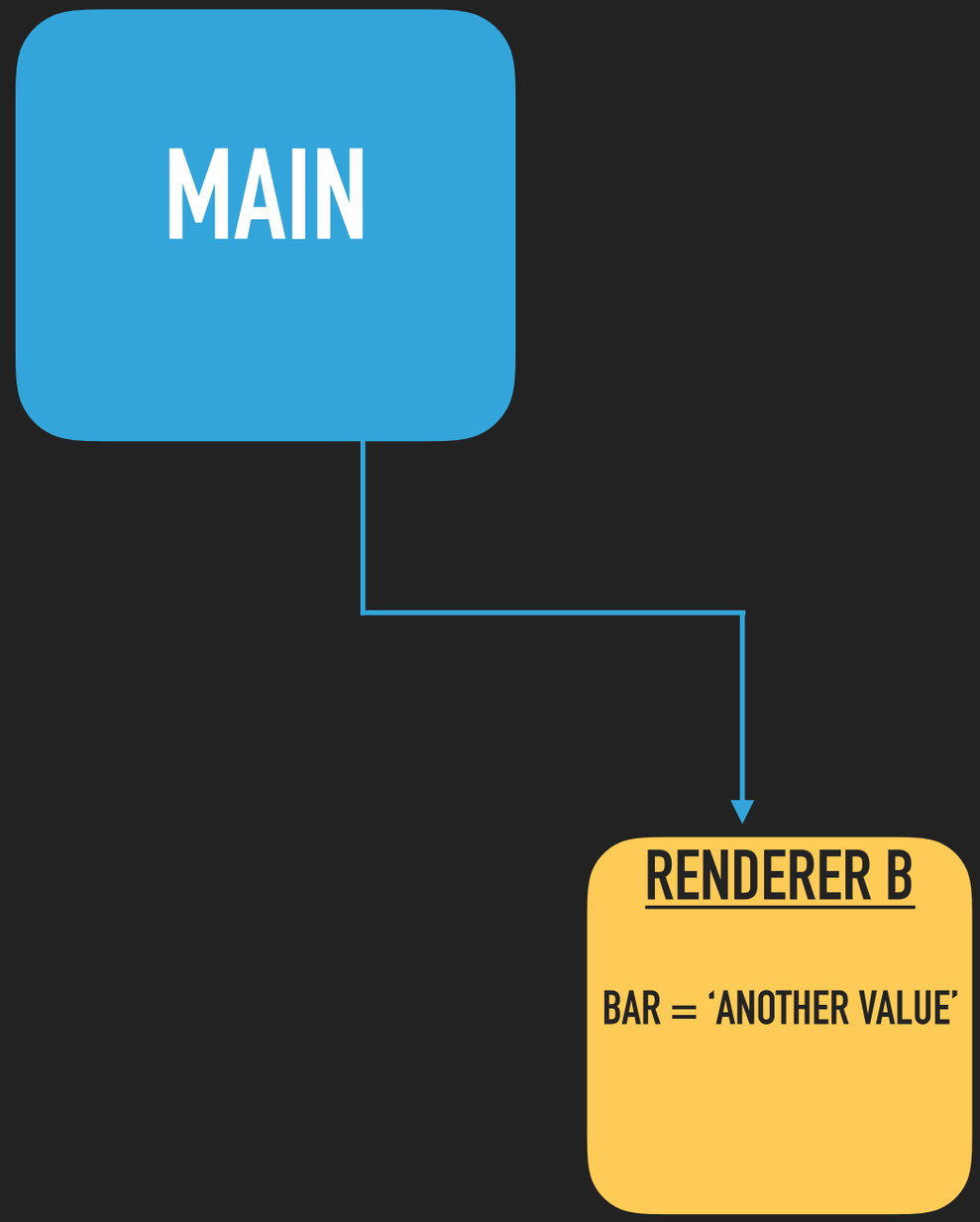
- Main and renderer processes communicate via extended EventEmitter classes
- ipcMain and ipcRenderer, respectively
- Render processes can also communicate with one another

PROBLEM: MAINTAINING A STATE

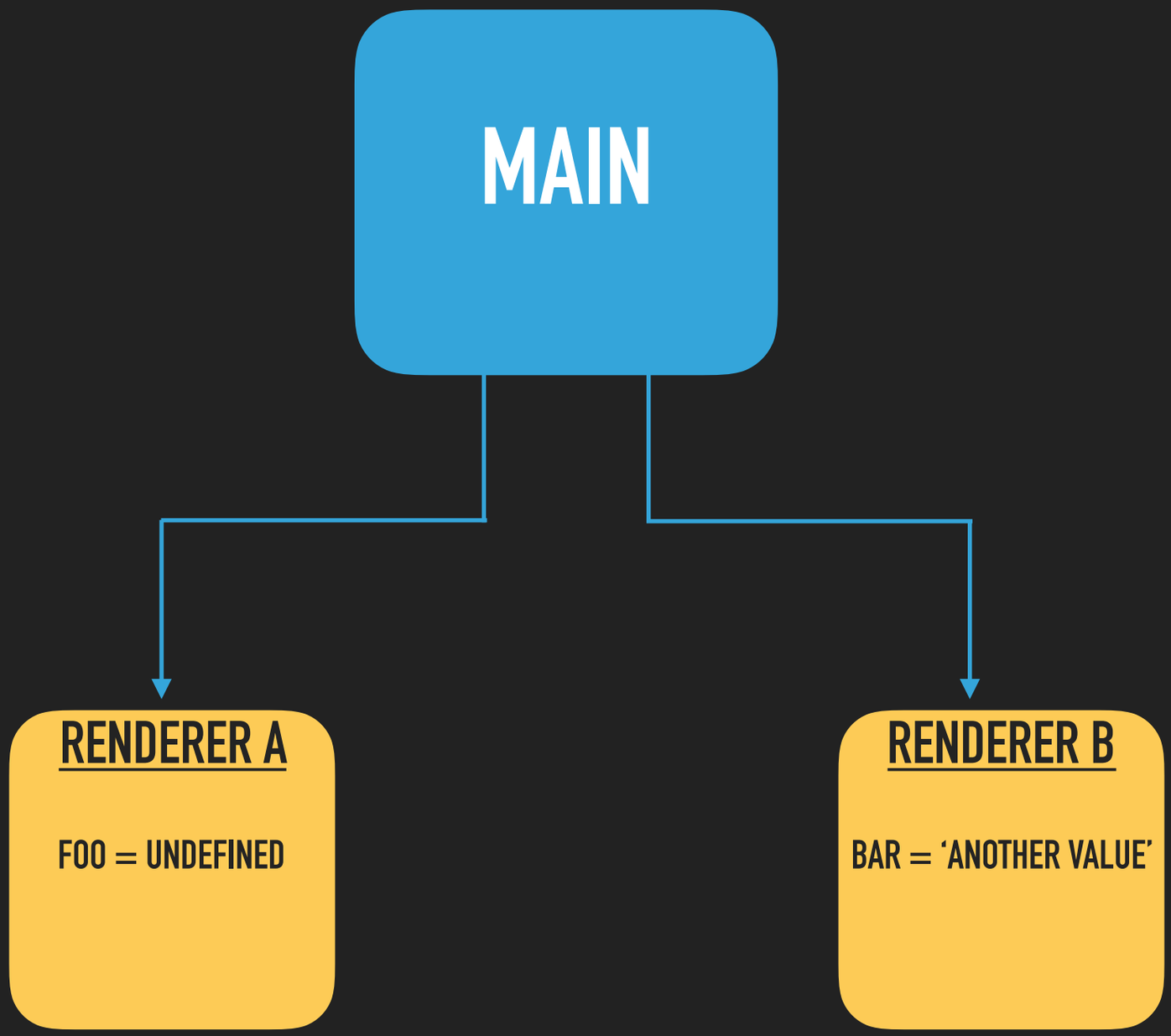
- Each process runs in its own thread. That means when the process ends, the values we've stored in the memory of that specific thread are deleted
- For the main process, this means termination of the app
- For the renderer process, this results in a loss of state



Two renderers - each with their own internal state



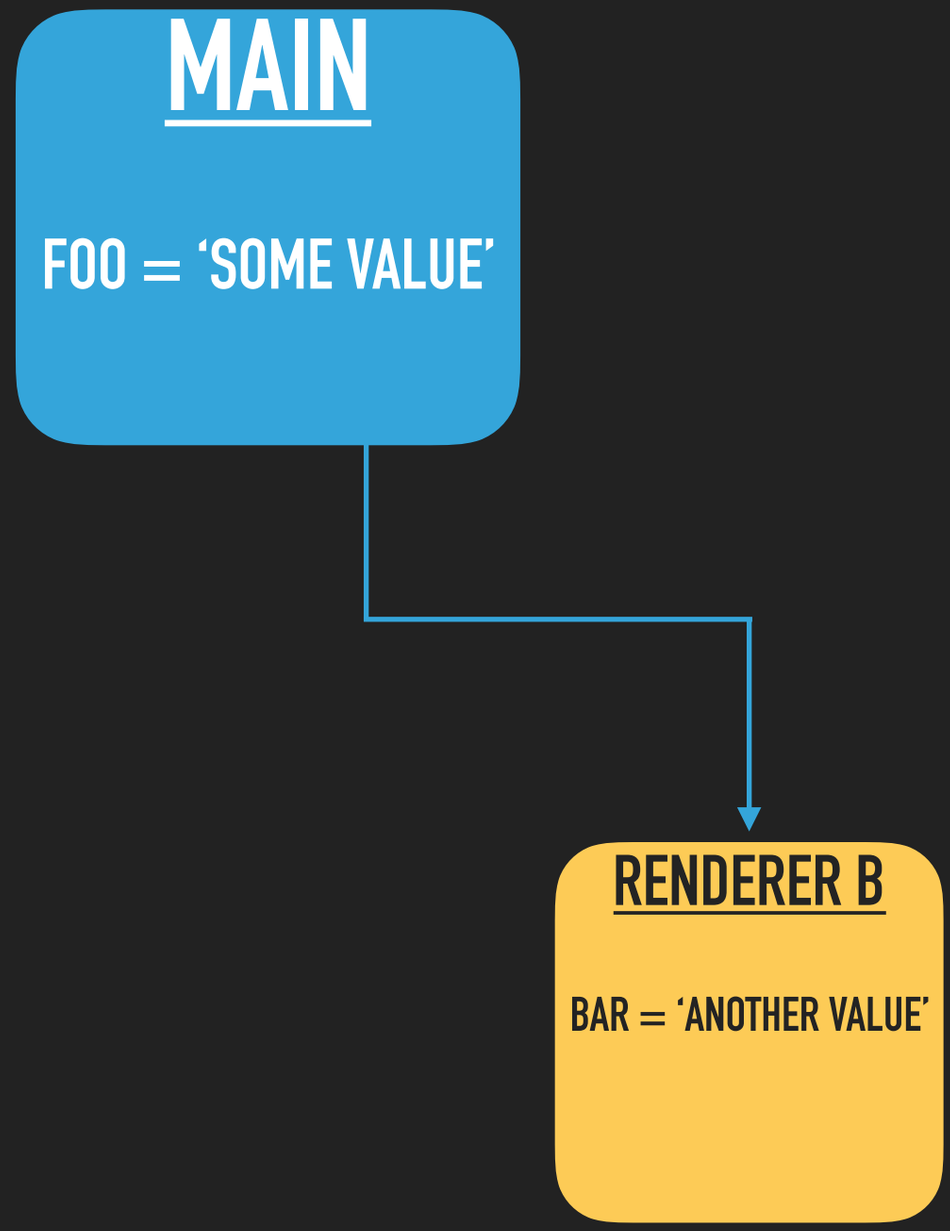
Renderer A is deleted



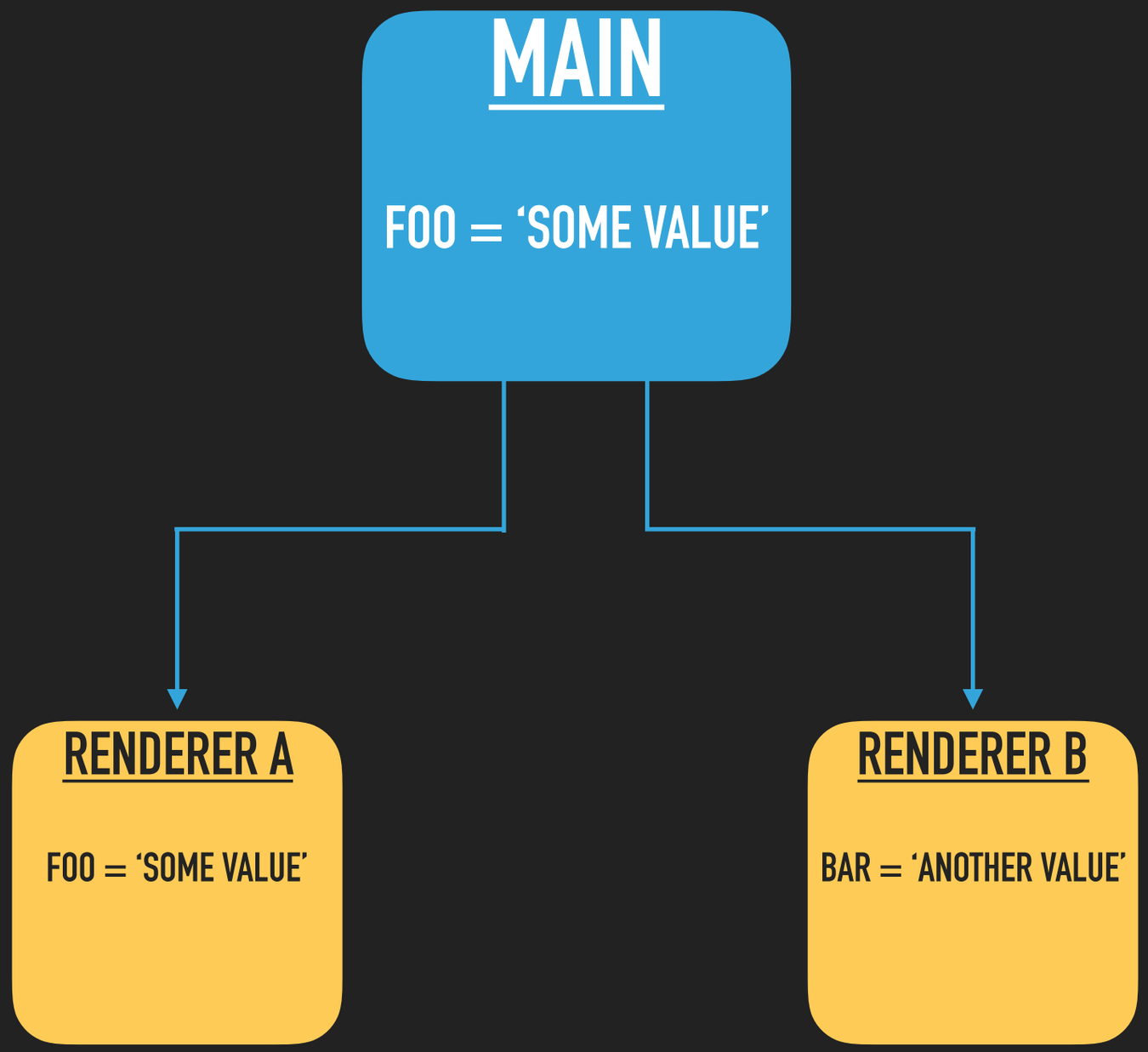
Renderer A created again - this time `foo` is undefined

SOLUTION: STORE STATE IN MAIN

- Keep any state that needs to be persisted in the main process
- Pass in the state into the render process upon creation



Main process holds a value of “some value” in `foo`



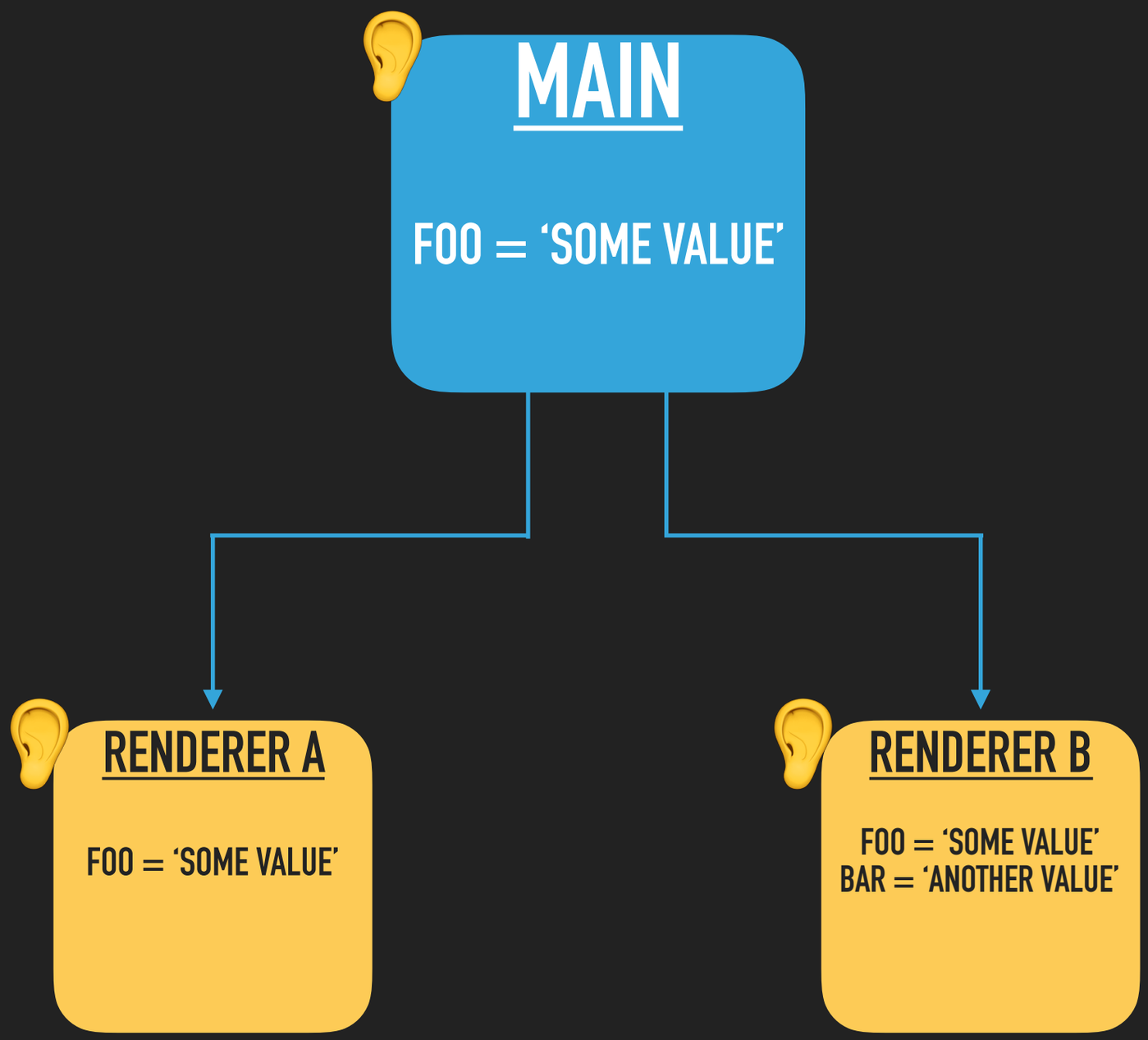
Renderer A spawned and is passed a value to set to `foo`

PROBLEM: SHARING A STATE BETWEEN RENDERERS

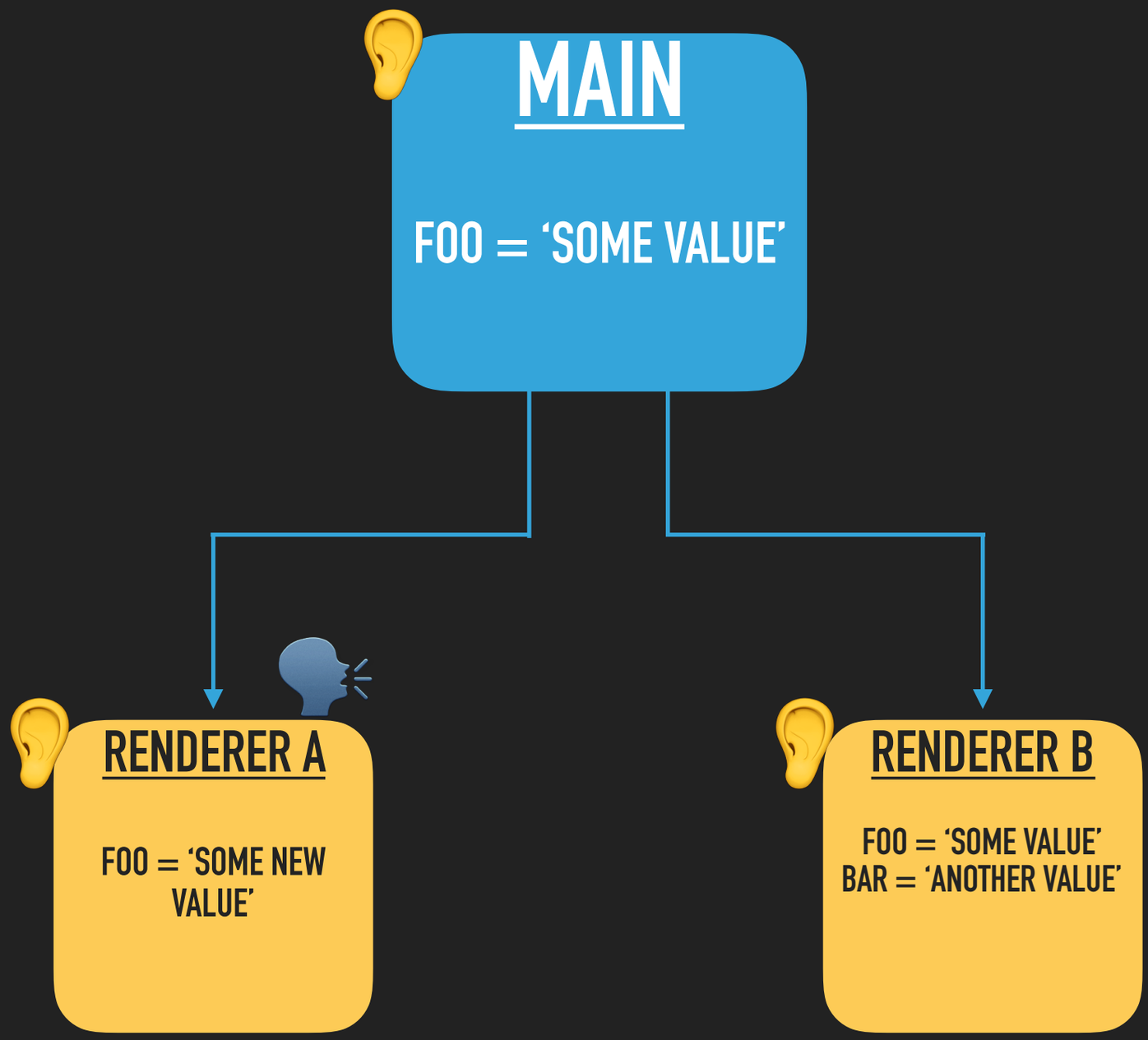
- Renderer A modifies a part of the main process state, but that updated value does not propagate to renderer B

SOLUTION: ANNOUNCE STATE CHANGES TO ALL RENDERERS

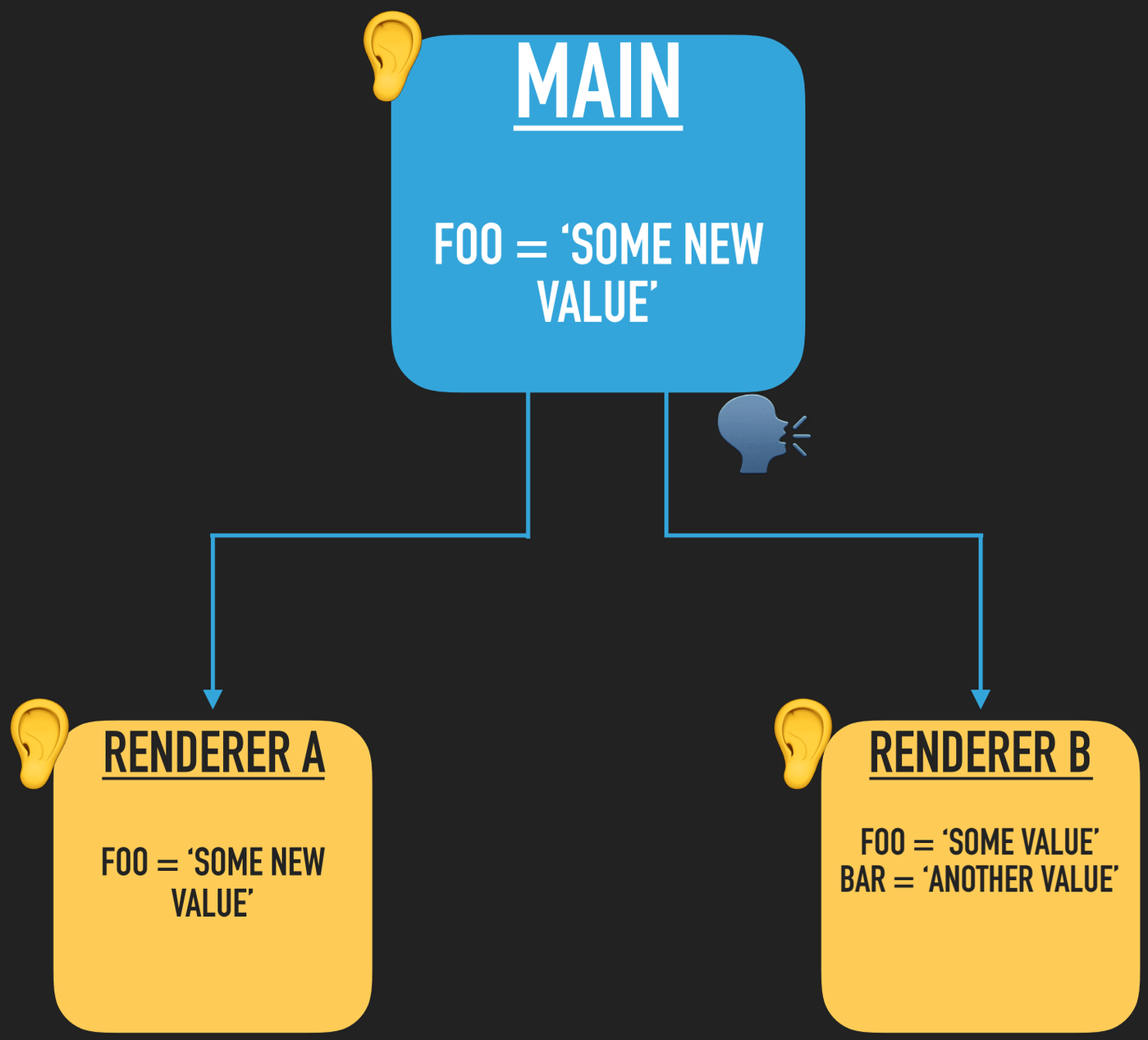
- When updated state is propagated to the main, emit an event to renderers and pass along that new state



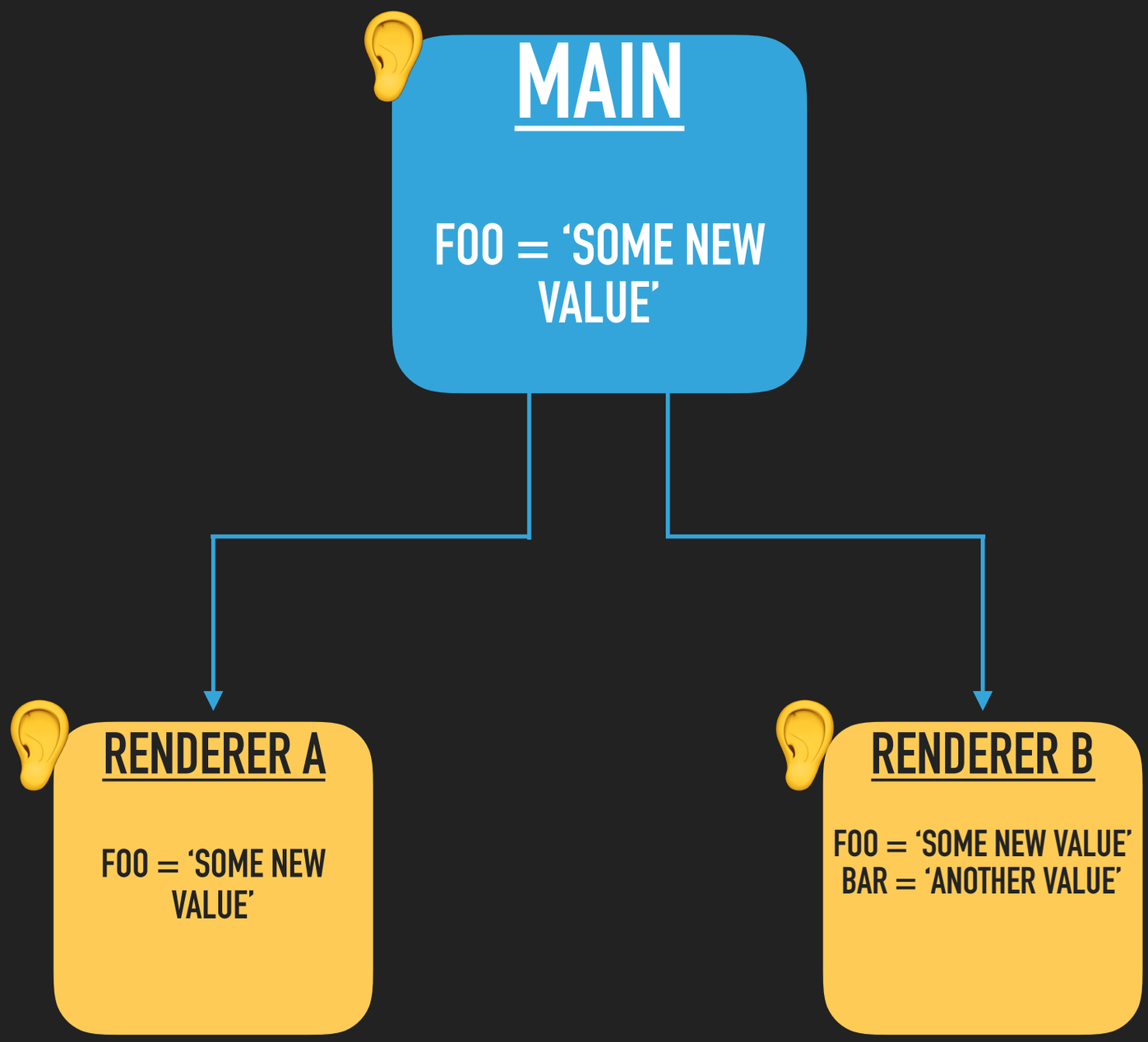
Listeners set up across all processes



Renderer A changes values of `foo` and emits to main



Main updates value of `foo` and emits to B



B updates values of foo

IMPLEMENTATION: SMOL-STORE

- A redux-like global store for electron apps
- Singleton that lives in the main process
- Modifications to the state are made through dispatching actions from renderer processes to the main
- Upon state updates, the renderer processes are pinged by the main with the updated state, and render changes to the UI if applicable

INITIALIZE (MAIN PROCESS)

```
initialize(state : Object, reducers : Object[function], [opts : Object])
```

- Takes an initial state, and object of reducer functions that map to properties on the state
- Calling initialize produces a side effect that creates a property called smolStore on the node global object
- Also initializes a dispatch method, which will pass along actions to your reducers

EXAMPLE REDUCER

```
module.exports = (state, action) ⇒ {
  switch (action.type) {
    case 'ADD_1':
      state += 1
      break
    case 'ADD_2':
      state += 2
      break
    case 'ADD_3':
      state += 3
      break
    default:
      state += action
  }
  return state
}
```

- Accepts FSA compliant actions
  - Must have a type property
  - May have a payload property
- State is not immutable - this has implications we'll explore later

DISPATCH (MAIN PROCESS)

```
dispatch(action : Object)
```

- A function called in the main process that sends an action along to all reducers
- Upon all reducers completing, a REFRESH event is emitted to any subscribed renderer processes, causing them to rerender

EXAMPLE DISPATCH

```
ipcMain.on('ADD_1', () ⇒ smolStore.dispatch({ type: 'ADD_1' }))
```

- Always placed in a callback for main process listeners
- In the above case, when any renderer process emits an ADD_1 message to the main, an action is dispatched to all reducers

SUBSCRIBE (RENDERER PROCESS)

```
subscribe(callback : Function)
```

- Emits a SUBSCRIBE event to the main process
  - Upon receiving, the main process adds the renderer to an array of renderers that are informed of updates to the state
- The callback is ran whenever a REFRESH event is heard
- In addition to subscribing when called, it will also drive the renderer to emit an UNSUBSCRIBE event when the unload event for the renderer process is fired off
  - This will remove the renderer from the array subscriptions in the main process

EXAMPLE SUBSCRIBE

```
subscribe(state ⇒ document.getElementById('count-container').innerText = state.count)
```

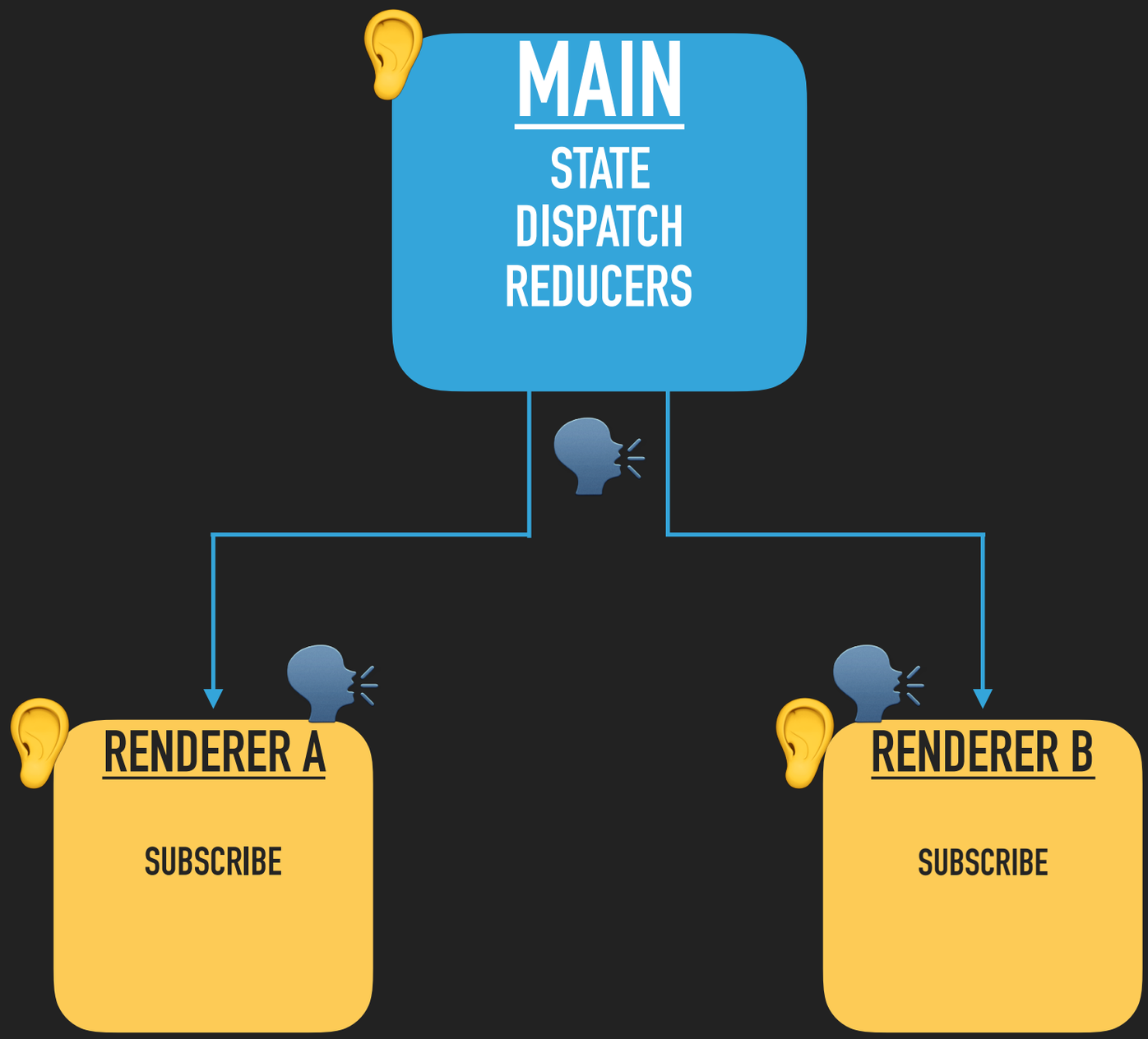
- Typically, a manipulation of the DOM occurs inside of the callback passed into render

EXAMPLE SUBSCRIBE (REACT)

```
class Provider extends React.Component {
  constructor() {
    super()
    this.state = {}
  }

  componentDidMount() {
    subscribe(state ⇒ this.setState({ ...state }))
  }

  render() {
    return React.cloneElement(this.props.children, { ...this.state })
  }
}
```



Layout of API

SIGNIFICANT DIFFERENCES FROM REDUX

- A mutable state will work, but will act as a bottleneck to performance at some point
  - The mapState function performs a shallow-compare of previous state and incoming - only rerendering the HOC's children when it detects a diff
  - With smolStore, no diff occurs and thus the components are always rerendered
- No ability to integrate middleware
  - No ability to shape the action or perform a side effect between dispatch and reduction