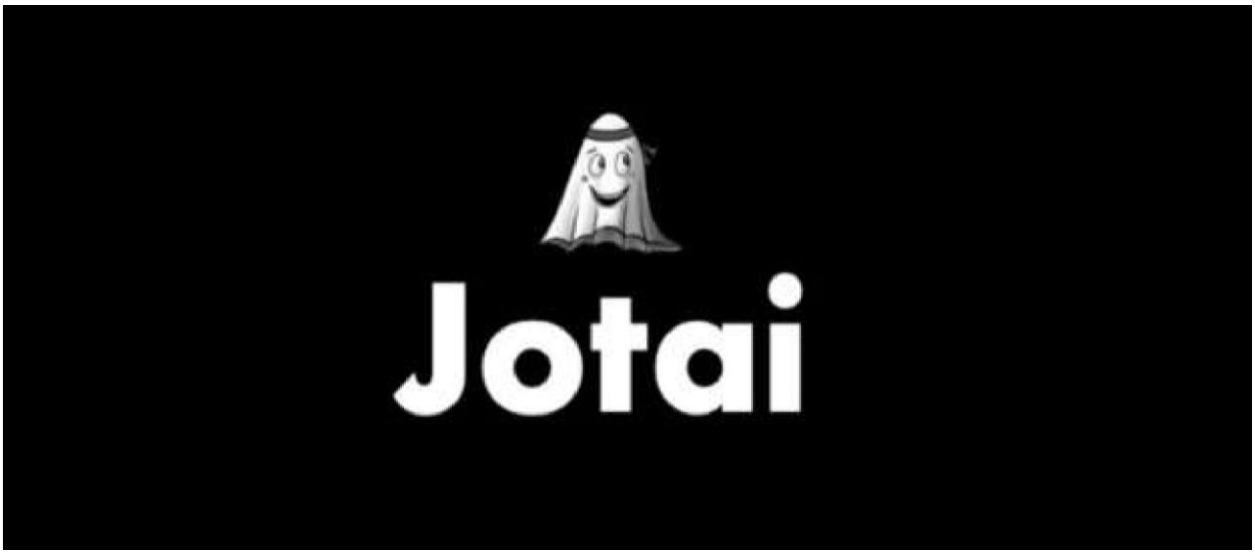
Writed by Ow-mahan-wO → https://github.com/Ow-mahan-wO



First of all, I want to give a basic explanation about stateManagement.

In our state managers, we have three types of architecture: flux, proxy, and atomic, each of which has a different structure and is not very similar. In flux architecture we have redux and zustand for example and in proxy architecture we have Mobx for example and in Atomic architecture we have Jotai or Recoil for example

What is Jotai ?

Jotai is a relatively new state management library for React which works with (atom). It's simple, but make no mistakes, it's a robust library. Jotai is based on the new Recoil pattern and library by Facebook.

Why we should learn Jotai ?!

Redux is often used in large-scale projects due to its ability to maintain states at a large scale. If we want to use state managers in a small

project, Redux doesn't seem reasonable and we have to use the context API. However, Jotai comes to our aid and makes managing states in the program easy with its atomic structure and is very fast and convenient. This is one reason to learn Jotai, and another reason may be that we have the opportunity to get acquainted with other architectures and state managers.

How to learn Jotai ?
We learn step by step with the official jotai document and its practice. Let's go to start.

Lesson 1:
at first step you most install jotai in your app:
npm command : npm install jotai
yarn command : yarn add jotai

and at second step we should set Provider for jotai to be able to access them:

```
./main.jsx

import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App.tsx'

import {Provider} from 'jotai'

ReactDOM.createRoot(document.getElementById('root')!).render(
  <React.StrictMode>
```

```
 <Provider>
  <App />
 </Provider>
 </React.StrictMode>,
)
```

Creating your first atom

Jotai atoms are small isolated pieces of state. Ideally, one atom contains very small data. Here's how you create your first atom.

```
import { atom } from 'jotai' ;
const counter = atom(0);
```

t is as simple to use as React's integrated **useState** hook, but all state is globally accessible.

```
const [count, setCounter] = useAtom(counter);
```

The atom we created is to be passed to **useState** hook with the help of jotai **useAtom** function, which returns an array, where the 1st element is the value of atom, and the 2nd element is a function used to set the value of the atom.

Jotai considers anything to be an atom so you can create any type of atom you want whether it is atom of objects, arrays, or nested objects.

```
const friendObj = atom({ name: "Shahin", online: false });
const cities = atom([ "Shiraz", "Tehran", "Kerman" ]);
const nestedObj = atom({ friend1: { name: "Korosh", age: 20 } })
```

Example Lesson 1 :

in this example we write a very small app with jotai so with Click in “Click for increased ” Button add 1 count to counter variable and more ….

```
import { atom, useAtom } from 'jotai';

const counter = atom(0);

export default function ExLesson1() {
 const [count, setCounter] = useAtom(counter);
 const setCountHandler = () => setCounter(count => count +=1);
 return (
  <div>
   <button onClick={ setCountHandler }>Click for Increased</button>
   <p>{count}</p>
  </div>
 )
```

As we learned in the above material, we made a number type variable using atoms and stored it inside a variable, and using useAtom, which has a structure similar to useState in React, we were able to update or use it.

Lesson 2 :

Persisting state value

In this lesson, we will take a look at how we can persist the state value to **localStorage** with jotai **atoms**. Persisting state to **localStorage** can be

challenging. You might want to persist the user's preferences or data for their next session.
Jotai atomWithStorage is a special kind of atom that automatically syncs the value provided to it with localstorage or sessionStorage, and picks the value upon the first load automatically. It's available in the jotai/utils module. To persist our theme atom simply create it with the atomWithStorage atom.

Note: In first Parameter we pass keyword for toggle and second Parameter we pass that status (true/false)

```
const theme = atomWithStorage('dark', false)
```

Example Lesson 2 :
In this Example we have a Toggler button for change new Theme in app So Click that button and to next Step we Refresh page and Theme changed and not be set to default theme . this is Amazing

```
import { useAtom } from 'jotai';
import { atomWithStorage } from 'jotai/utils';

const theme = atomWithStorage('dark', false);

export default function ExLesson2() {
  const [Theme, setTheme] = useAtom(theme);
  const handleThemeClick = () => setTheme(!Theme);
  return (
    <div className={Theme? 'dark': 'light'}>
      <h1>This is a theme switcher</h1>
                <button     onClick={handleThemeClick}>{Theme?     'DARK':
'LIGHT'}</button>
    </div>
```

```
)
```

Lesson 3 :
Read Only atoms:
Readonly atoms are used to read the value of the other atoms. You can't set or change their value directly because these atoms rely on their parent atoms.

```
const textAtom = atom("readonly");
const uppercase = atom((get) => get(textAtom).toUpperCase());
```

you can with (get) method get a value of atom but not available method for set value on that atom its call (Read Only Atoms)

```
const firstName = atom("Amir");
const lastName = atom("Jamshidi");
const fullName = atom((get) => get(firstName) + " " + get(lastName));
```

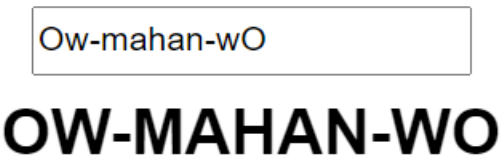
you can get one or two and more values of atoms and use it same Example above.

Example Lesson 3:

```
import { atom, useAtom } from "jotai";

const text = atom("Ow-mahan-wO");
const Uppercase = atom((get) => get(text).toUpperCase());

export default function ExLesson3() {
  const [lowercaseText, setLowercaseText] = useAtom(text);
```

```
  const [uppercaseText] = useAtom(Uppercase);
  const handleChange = (e) => setLowercaseText(e.target.value);
  return (
    <div className="app">
      <input value={lowercaseText} onChange={handleChange} />
      <h1>{uppercaseText}</h1>
    </div>
  );
}
```

we have this output in the end :



Awsome feature:

You can do more than just simply read the value of other atoms like (filter) and sorted out them or (map) over the values of the parent atom. And this is the beauty of it, Jotai gracefully lets you create dumb atoms derivated from even more dumb atoms.Here is a example of getting the list of all online and offline friends:

```
const friendsStatus = atom([
  { name: "Reza", online: false },
  { name: "Shayan", online: true },
  { name: "Mahan", online: false },
]);
```

```
const onlineFriends = atom((get) =>
  get(friendsStatus).filter((item) => item.online)
);
const offlineFriends = atom((get) =>
  get(friendsStatus).filter((item) => !item.online)
);
```

it is greate and easy!!!

Lesson 4:

Read Write atoms :

These atoms are the combination of both read-only and write-only atoms.

```
const count = atom(1);
export const readWriteAtom = atom((get) => get(count),
(get, set) => {
   set(count, get(count) + 1);
  },
);
```

The first parameter is for reading and the second is for modifying the atom value. Since the readWriteAtom is capable to read and set the original atom value, so we can only export readWriteAtom atom and can hide the original atom in a smaller scope.

Example Lesson 4:

```
import { atom, useAtom } from 'jotai';

const counter = atom(1);
export default function ExLesson4() {
```

```
const readWriteAtom = atom((get) => get(counter),
(get, set) => {
   set(counter, get(counter) + 1);
 },
);
 const [count,setCount]=useAtom(readWriteAtom)
 return (
  <div>
    <h1>{count}</h1>
    <button onClick={setCount}>Click</button>
  </div>
 )
}
```

In this example we have a button and h1 element that show count and when clicked button, count increased 1 and update element value:

Lessson 5

Atom Creators:

An atom creator means simply a function that returns an atom or a set of atoms. It's just a function and it's not some features that the library provides, but it's an important pattern to make a fairly complex use case. This avoids the boilerplate of having to set up another atom just to update the state of the first.

```
const createCountIncreasedAtoms = (initialValue) => {
  const InitialAtom = atom(initialValue)
  const ValueAtom = atom((get) => get(InitialAtom))
  const IncreamentAtom = atom(null, (get, set) => set(InitialAtom, (n) => n + 1))
  return [ValueAtom, IncreamentAtom]
}

const [TestAtom1, TestIncAtom1] = createCountIncreasedAtoms(0)
const [TestAtom2, TestIncAtom2] = createCountIncreasedAtoms(0)
```

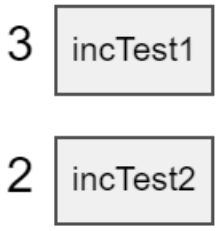
Consider this case that use it of write atoms that we learned in previous Lessons:

```
const TestAtom1 = atom(0);
const TestAtom2 = atom(0);
const TestincAtom1 = atom(null, (get, set) => {
   set(TestAtom1, n => n + 1);
});
const TestincAtom2 = atom(null, (get, set) => {
   set(TestAtom2, n => n + 1);
});
```

Although you can attach the suitable actions to the setter of the respective atom, but this also increases boilerplate code when there are more atoms in your code.
So simply replace this with the atom creators function.
Example Lesson 5:

```
import { atom, useAtom } from 'jotai'

const createCountIncreasedAtoms = (initialValue) => {
  const InitialAtom = atom(initialValue)
  const ValueAtom = atom((get) => get(InitialAtom))
  const IncreamentAtom = atom(null, (get, set) => set(InitialAtom, (n) => n + 1))
```

```
  return [ValueAtom, IncreamentAtom]
}

const [TestAtom1, TestIncAtom1] = createCountIncreasedAtoms (0)
const [TestAtom2, TestIncAtom2] = createCountIncreasedAtoms (0)

function ExLesson5() {
  const [TestCount1] = useAtom(TestAtom1)
  const [, incTest1] = useAtom(TestIncAtom1)
  const [TestCount2] = useAtom(TestAtom2)
  const [, incTest2] = useAtom(TestIncAtom2)

  const onClick1 = () => {
    incTest1()
  }
  const onClick2 = () => {
    incTest2()
  }

  return (
    <>
      <div>
        <span>{TestCount1}</span>
        <button onClick={onClick1}>incTest1</button>
      </div>
      <div>
        <span>{TestCount2}</span>
        <button onClick={onClick2}>incTest2</button>
      </div>
    </>
  )
}
```



in this example we have two atoms that should be update similar each other . we write a atom creator and handle proccess in that and make it reusable , with pass parameter to custome value in atom creator function.

Lesson 6

Async Read Atoms:

Using async atoms, you gain access to real-world data while still managing them directly from your atoms and with incredible ease.
We separate async atoms in two main categories , Async read atoms and Async write atoms.
Like example below :

```
const counter = atom(0);
const asyncAtom = atom(async (get) => get(counter) * 5);
```

Jotai is inherently leveraging Suspense to handle asynchronous flows.

```
<Suspense fallback={<span>loading...</span>}>
    <AsyncComponent />
</Suspense>
```

But there is a more jotai way of doing this with the loadable api present in jotai/utils. By simply wrapping the atom in loadable util and it returns the value with one of the three states: loading, hasData and hasError.

```
import { loadable } from "jotai/utils"
import { atom, useAtom } from 'jotai'

const countAtom = atom(0);
const asyncAtom = atom(async (get) => get(countAtom));
const loadableAtom = loadable(asyncAtom)
const AsyncComponent = () => {
  const [value] = useAtom(loadableAtom)
  if (value.state === 'hasError') return <div>{value.error}</div>
  if (value.state === 'loading') {
    return <div>Loading...</div>
  }
  return <div>Value: {value.data}</div>
}
```

Example Lessson 6 :

```
import { atom, useAtom } from 'jotai';
import { Suspense } from 'react'

const counter = atom(1);
const asyncAtom = atom(async (get) => get(counter) * 5);

function AsyncComponent() {
  const [asyncCount] = useAtom(asyncAtom);
  return (
    <div className="app">
      <h1>{asyncCount}</h1>
    </div>
  )
}

export default function ExLesson6() {
  return (
```

```
  <Suspense fallback={<span>loading...</span>}>
    <AsyncComponent />
  </Suspense>
 )
}
```

In example above we have a count atom that read with async and do it count * 5

And in suspense we pass a fallback props with value that wana show in loading new value .

Lesson 7

Async Write Atoms

In Async write atoms the write function of atom returns a promise.

We pass two parameter in atom , first parameter all the time like write atoms , are null and second parameter are callback fn (set , get) with this the difference we write async keyword before it.

Like this :

```
const counter = atom(0);
const asyncAtom = atom(null, async (set, get) => {
  // await something
  set(counter, get(counter) + 1);
});
```

This is important that , An important take here is that async write function does not trigger the Suspense

But an interesting pattern that can be achieved with Jotai is switching from async to sync to trigger suspending when wanted.

```
const request = async () => fetch('https://api/...').then((res) =>
res.json())
const baseAtom = atom(0)
const Component = () => {
  const [value, setValue] = useAtom(baseAtom)
  const handleClick = () => {
    setValue(request()) // Will suspend until request resolves
  }
  // ...
}
```

Example Lesson 7 :

In this example we have two component (Suspence , AsyncComponent)
In AsyncComponent we have a counter atom that with Click in inc Button increased and fetch data of JsonPlaceholder api (fake api) and show it.
And in Suspence we have a default object with data (id,title,etc.) and write Async request for get data in api and so on we take data and set in useAtom till we can use them .
And set Click event on button, until the click on button we set request data on atom .

```
./App.js

import { atom, useAtom } from 'jotai';
import Suspense from './Suspense.js';

const counter = atom(1);
const asyncAtom = atom(null, async (get, set) => {
   await fetch('https://jsonplaceholder.typicode.com/todos/');
   set(counter, get(counter) + 1);
});
```

```
function AsyncComponent() {
  const [count] = useAtom(counter);
  const [, incCounter] = useAtom(asyncAtom);
  return (
    <div className="app">
      <h1>{count}</h1>
      <button onClick={incCounter}>inc</button>
    </div>
  )
}
export default function ExLesson7() {
   return (
    <div>
      <AsyncComponent />
      <Suspense />
    </div>
  )
}
```

./Suspence.js

```
import { atom, useAtom } from 'jotai';
import { Suspense } from 'react'

const todo = {
    id: 0,
    title: 'how to learn jotai',
    completed: true
};

const request = async () => (
    fetch('https://jsonplaceholder.typicode.com/todos/5')
        .then((res) => res.json())
)
const todoAtom = atom(todo);

function Component() {
```

```
  const [todoGoal, setGoal] = useAtom(todoAtom);
  const handleClick = () => {
    setGoal(request());
  }
  return (
    <div>
      <p>Todays Goal: {todoGoal.title}</p>
      <button onClick={handleClick}>New Goal</button>
    </div>
  )
}

export default function AsyncSuspense() {
  return (
    <div>
      <Suspense fallback={<span>loading...</span>}>
        <Component />
      </Suspense>
    </div>
  )
}
```

Lesson 8

This is an overview of the atom creators/hooks utilities that can be found under jotai/utils. We already covered atomWithStorage and loadable API in previous lessons.
And we are going to learn a few more together.

1 -atomWithReset Creates an atom that could be reset to its initialValue with useResetAtom hook. It works exactly the same way as primitive atom would, but you are also able to set it to a special value RESET.

```
import { atomWithReset } from 'jotai/utils'

const counter = atomWithReset(1)
```

as easy as that.
So going write a example :

```
import { useAtom } from 'jotai';
import { atomWithReset, useResetAtom } from 'jotai/utils';

const counter = atomWithReset(1);

export default function Counter() {
 const [count, setCount] = useAtom(counter);
 const reset = useResetAtom(counter);

 const inc = () => setCount(c => c * 2);

 return (
  <div className="counter">
   <p className="util">1. atomWithReset</p>
   <p>{count}</p>
   <div>
    <button onClick={inc}>Inc</button>
```

```
      <button onClick={reset}>Reset Count</button>
    </div>
   </div>
  );
}
```

In this example we have two button of names Inc(that when click on them increased count atom) and Reset Count(that when click on them reset count atom to initial value with atomWithReset()).

2.selectAtom This function creates a derived atom whose value is a function of the original atom's value, determined by selector. The selector function runs whenever the original atom changes; it updates the derived atom only if equalityFn reports that the derived value has changed. By default, equalityFn is reference equality, but you can supply your favorite deep-equals function to stabilize the derived value where necessary.

```
const defaultPerson = {
   name: {
    first: 'Mahan',
    last: 'Heydari',
   },
   birth: {
    year: 2005,
    month:'nov',
    day: 19,
   }
 }

 // Original atom.
 const personAtom = atom(defaultPerson)
 const nameAtom = selectAtom(personAtom, (person) => person.name,
deepEqual)
```

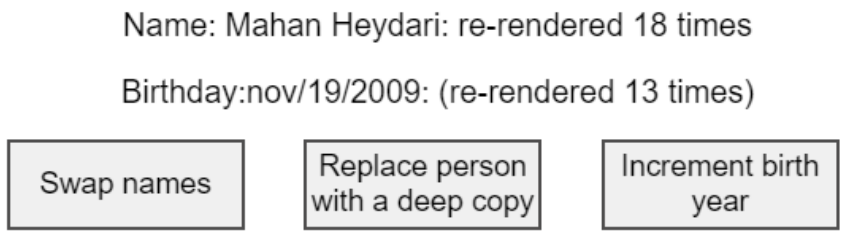
So going write a example :

```
import { atom, Provider, useAtom } from "jotai";
import { selectAtom } from "jotai/utils";
import { useRef, useEffect } from "react";
import { isEqual } from 'lodash-es';

const defaultPersonData = {
  name: {
    first: "Mahan",
    last: "Heydari"
  },
  birth: {
    year: 2005,
    month: "nov",
    day: 19,
    time: {
      hour: 0,
      minute: 0
    }
  }
};

// Original atom.
const personDataAtom = atom(defaultPersonData);

const nameAtom = selectAtom(personDataAtom, (person) => person.name);
const birthAtom = selectAtom(personDataAtom, (person) => person.birth,
isEqual);

const useCommitCount = () => {
  const rerenderCountRef = useRef(0);
  useEffect(() => {
    rerenderCountRef.current += 1;
  });
  return rerenderCountRef.current;
};
```

```
// Rerenders when nameAtom changes.
const DisplayName = () => {
  const [name] = useAtom(nameAtom);
  const n = useCommitCount();
  return (
    <div>
      Name: {name.first} {name.last}: re-rendered {n} times
    </div>
  );
};

// Re-renders when birthAtom changes.
const DisplayBirthday = () => {
  const [birth] = useAtom(birthAtom);
  const n = useCommitCount();
  return (
    <div>
      Birthday:
      {birth.month}/{birth.day}/{birth.year}: (re-rendered {n} times)
    </div>
  );
};

const SwapNames = () => {
  const [person, setPerson] = useAtom(personDataAtom);
  const handleChange = () => {
    setPerson({
      ...person,
      name: { first: person.name.last, last: person.name.first }
    });
  };
  return <button onClick={handleChange}>Swap names</button>;
};

const CopyPerson = () => {
  const [person, setPerson] = useAtom(personDataAtom);
```

```
  const handleClick = () => {
    setPerson({
      name: { first: person.name.first, last: person.name.last },
      birth: {
        year: person.birth.year,
        month: person.birth.month,
        day: person.birth.day,
        time: {
          hour: person.birth.time.hour,
          minute: person.birth.time.minute
        }
      }
    });
  };
  return <button onClick={handleClick}>Replace person with a deep
copy</button>;
};

// Changes birth year, triggering a change to birthAtom, but not nameAtom.
const IncrementBirthYear = () => {
  const [person, setPerson] = useAtom(personDataAtom);
  const handleClick = () => {
    setPerson({
      name: person.name,
      birth: { ...person.birth, year: person.birth.year + 1 }
    });
  };
  return <button onClick={handleClick}>Increment birth year</button>;
};

export default function App() {
  return (
    <div className="selectAtom">
      <p className="util">2. selectAtom</p>
      <Provider>
        <DisplayName />
        <DisplayBirthday />
```

```
    <div className="btn-group">
      <SwapNames />
      <CopyPerson />
      <IncrementBirthYear />
    </div>
   </Provider>
  </div>
 );
}
```



3-splitAtoms: The splitAtom utility is useful for when you want to get an atom for each element in a list. It works for read and write atoms that contain a list. When used on such an atom, it returns an atom which itself contains a list of atoms, each corresponding to the respective item in the original list.

Additionally, the atom returned by splitAtom contains a dispatch function in the write direction , this is useful for when you want a simple way to modify the original atom with actions such as remove, insert, and move.

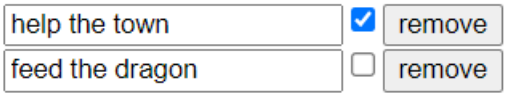
See the below example for usage.

```
import { Provider, atom, useAtom, PrimitiveAtom } from 'jotai'
import { splitAtom } from 'jotai/utils'
import './styles.css'

// fake datas
const initialState = [
  {
    task: 'Go to Park,
    done: false,
  },
  {
    task: 'Check the Emails',
    done: false,
  },
]

// data atom
const todosAtom = atom(initialState)
const todoAtomsAtom = splitAtom(todosAtom)

const TodoItem = ({
  todoAtom,
  remove,
}: {
  todoAtom: PrimitiveAtom
  remove: () => void
}) => {
  const [todo, setTodo] = useAtom(todoAtom)
  return (
    <div>
      <input
        value={todo.task}
        onChange={(e) => {
          setTodo((oldValue) => ({ ...oldValue, task: e.target.value }))
        }}
      />
      <input
```

```
      type="checkbox"
      checked={todo.done}
      onChange={() => {
        setTodo((oldValue) => ({ ...oldValue, done: !oldValue.done }))
      }}
     />
     <button onClick={remove}>remove</button>
   </div>
  )
}

const TodoList = () => {
  const [todoAtoms, dispatch] = useAtom(todoAtomsAtom)
  return (
    <ul>
     {todoAtoms.map((todoAtom) => (
       <TodoItem
         todoAtom={todoAtom}
         remove={() => dispatch({ type: 'remove', atom: todoAtom })}
       />
     ))}
    </ul>
  )
}

const App = () => (
  <Provider>
    <TodoList />
  </Provider>
)

export default App
```

in this example we have a mini todolist similar finally project for complete this Article .

we have a default data and with SplitAtoms As we have learned get items in default data and create component for that show on . we create for now TodoItem component.
in TodoItem Component we have a checkbox for check done this todo or not , and we have a input for update todo value , and a delete button for remove todo in list .



We learned some useful and important utilities in this lesson.

Lesson 9

Integrations

Updating the state with jotai is simple with the provided set function but things can go complex and requires some extra effort with the nested object states as you have to copy the state at each level with the spread operator ... like so,

```
...
setAtomTodo(state => {
  const deepCopyState = {
    ...state,
     todo: {
       ...state.todo,
       person: {
         ...state.todo.person,
         title: {
           ...state.todo.person.title,
```

```
        goal: "new title"
       }
      }
     }
   }
 return deepCopyState;
});
```

This is a very naive method and there may be a higher chance that you make some mistakes while updating the state like this.
To make our life easy we can take advantage of jotai 3rd party library's support. Jotai officially supports Immer, Optics, Zustand , Redux, tRPC, and various other 3rd party integrations.
Let's see how we can use immer to directly mutate the state, You have to install immer and jotai-immer to use this feature.

```
npm install immer jotai-immer
```

Create a new atom with atomWithImmer.

```
import { atomWithImmer } from 'jotai-immer';

const immerAtom = atomWithImmer(todo);
...
const updateTodo = () => {
 setAtomTodo(immerTodo => {
  // directly mutating the state with immer
  immerTodo.todo.person.title = "new title";
  return immerTodo;
 });
}
```

atomWithImmer creates a new atom similar to the regular atom with a different writeFunction. In this bundle, we don't have read-only atoms,

because the point of these functions is the immer produce(mutability) function.

Example Lesson 9 :

```
import { useAtom } from 'jotai';
import { atomWithImmer } from 'jotai-immer';

const todo = {
  todo: {
    person: {
      name: "Mahan",
       title: {
         goal: "old todo"
       },
    }
  }
};
const immerAtom = atomWithImmer(todo);

export default function ExLesson9() {
  const [todoAtom, setAtomTodo] = useAtom(immerAtom);
  const updateTodo = () => {
    setAtomTodo(state => {
       state.todo.person.title.goal = "Update title";
       return state;
    });
  }
  return (
    <div>
      <h3>Name: {todoAtom.todo.person.name}</h3>
      <p>Todo: {todoAtom.todo.person.title.goal}</p>
      <button onClick={updateTodo}>Update Todo</button>
    </div>
  )
}
```

And let’s go to write finall TodoList Project with jotai.