React-similar framework "FIMI"

Framework “FIMI” works with the jsx standard -> you can return your HTML structure directly from the function. Babel compiler is used for "translation", more information here: https://en.wikipedia.org/wiki/Babel_(transcompiler)

Framework is based on the virtualDOM:
Virtual DOM is like a lightweight copy of the actual DOM (a virtual representation of the DOM). So for every object that exists in the original DOM, there is an object for that in Virtual DOM. It is exactly the same, but it does not have the power to directly change the layout of the document. Manipulating DOM is slow, but manipulating Virtual DOM is fast as nothing gets drawn on the screen.

When anything new is added to the application, a virtual DOM is created and it is represented as a tree. Each element in the application is a node in this tree. So, whenever there is a change in the state of any element, a new Virtual DOM tree is created. This new Virtual DOM tree is then compared with the previous Virtual DOM tree and makes a note of the changes. After this, it finds the best possible ways to make these changes to the real DOM. Now only the updated elements will get rendered on the page again.

HOW IT WORKS:

To start new project you have to have following structure:

Project_folder:
- > “dist” folder with your index.html file in it
- > “framework” folder with all framework files in it
- > “src” folder with your index.js file in it
- > all accompanying set up files from our framework project:
  - .babelrc - transforms jsx to regular js using our “FIMI” Vdom createElement
  - .package.json
  - webpack.config.js - creation of bundle.js
  - docker-compose.yml
  - start.sh

When you have this structure created

1) npm install -> to install all dependencies
2) npm run build-prod -> to create all production build
3) you can open index.html file with “Live Server”

HOW TO USE

index.html example:

index.html

Documentarion > dist > index.html > ...

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>EXAMPLE</title>
</head>
<body>
    <div id="app"></div>
    <script src="bundle.js"></script>
</body>
</html>
```

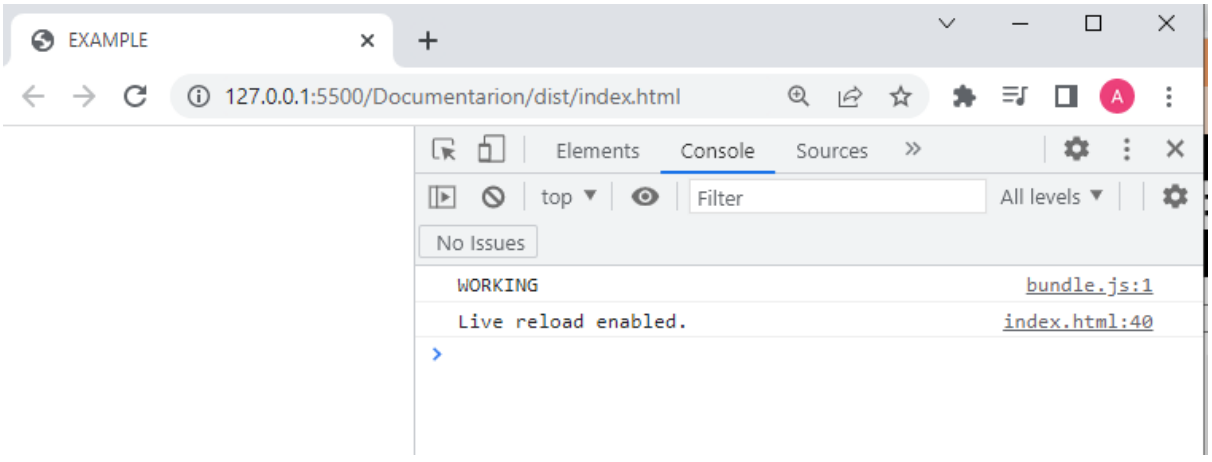
src>index.js:

index.js

Documentarion > src > index.js

```
console.log("WORKING")
```

Now you can open page on localhost and see the console:



ADDING ELEMENTS TO PAGE:

To be able to use the framework you have to import render function and VDom function:

```
import VDom from "../framework/Vdom";
import {render} from "../framework"
```

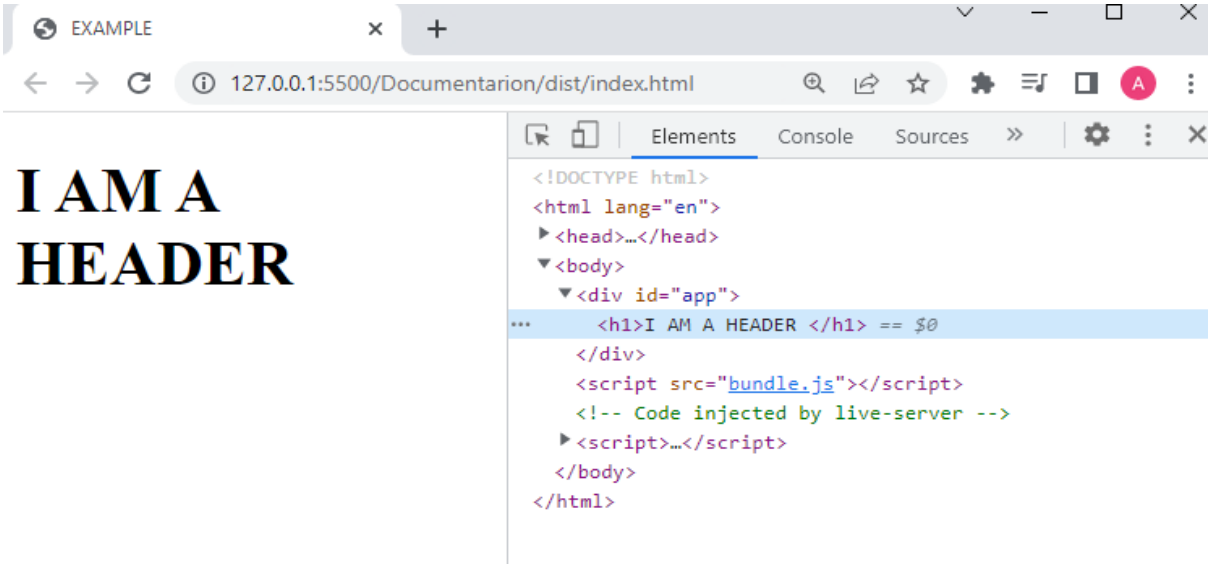
Babel uses VDom (createElement) function to compile the jsx format.
Render function needs to apply changes (add virtual DOM changer to real DOM tree)

To create an element you need to create a function that returns an HTML element and render(add) it to the page.

index.js

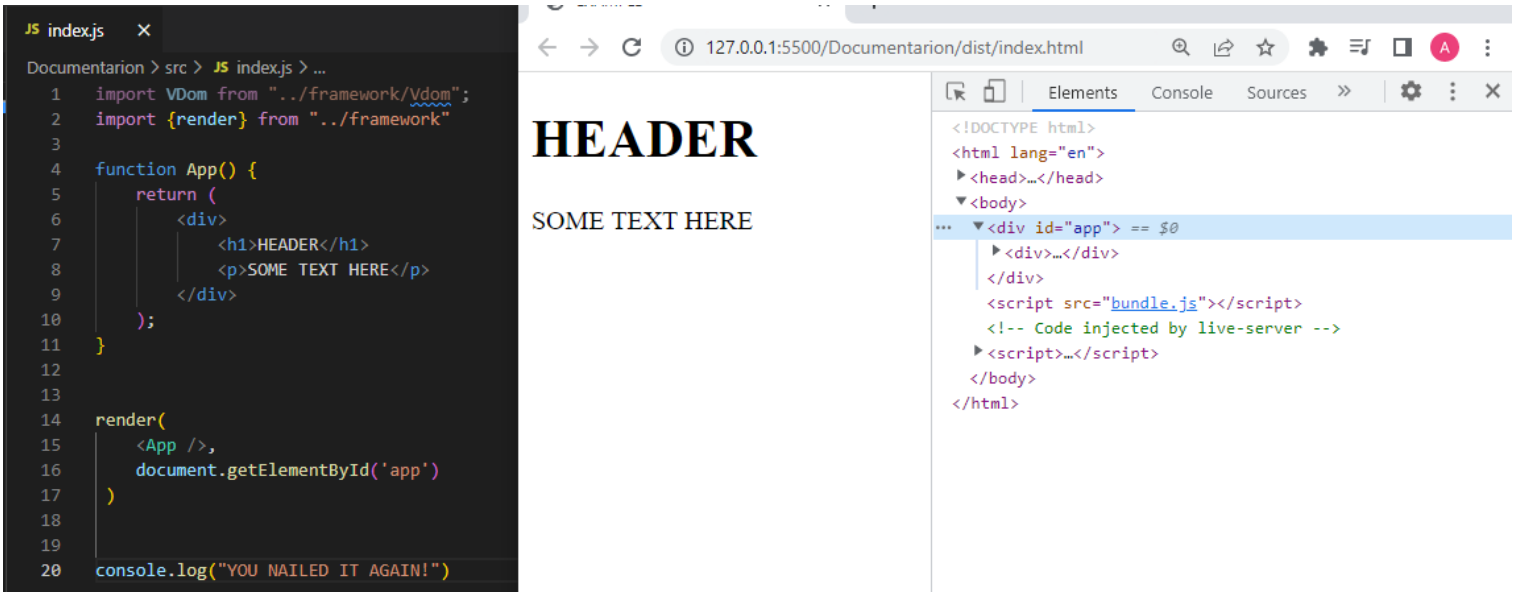
```
Documentarion > src > JS index.js > ...
1   import VDom from "../framework/Vdom";
2   import {render} from "../framework"
3
4   function App() {
5       return (
6           <h1>I AM A HEADER </h1>
7       );
8   }
9
10  render(
11      <App />,
12      document.getElementById('app')
13  )
14
15
16  console.log("YOU NAILED IT!")
```

OUTPUT:
element H1 is added to div with id "app"



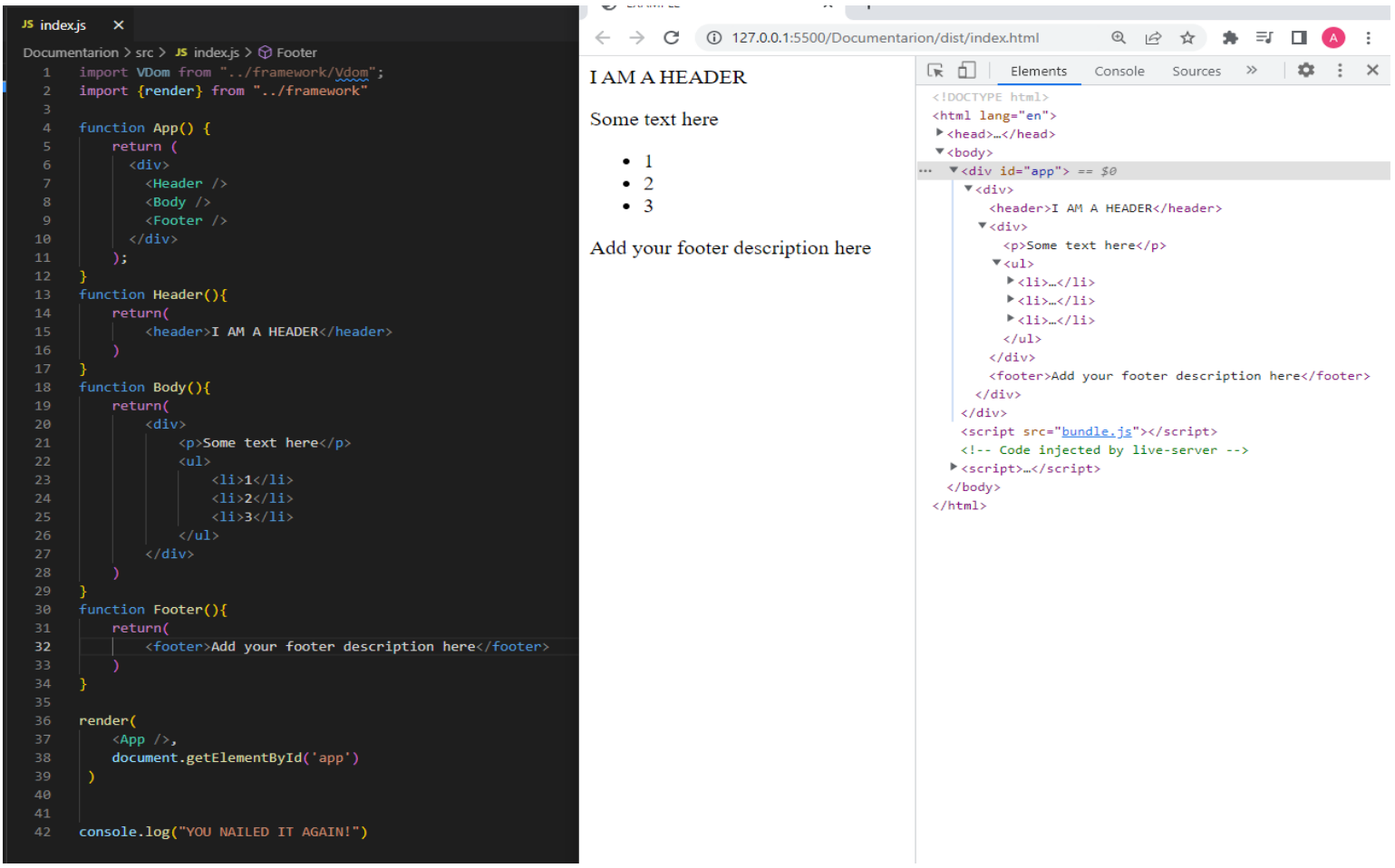
EX2:

If more the 1 elements need to be added, then they have to be "wrapped", as ONLY 1 element can be rendered:



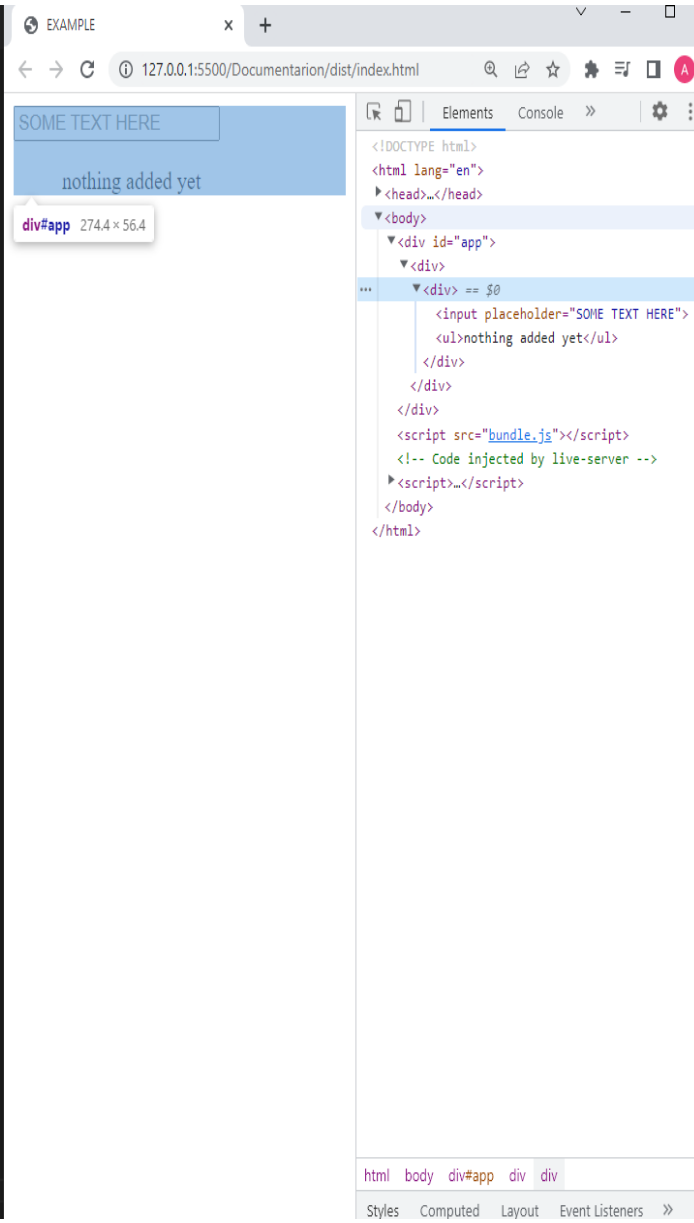
EX3:

Like in React you can divide elements to parts:



Now let's make some page with input and make the page to re-render “onEnter”

```
import VDom from "../framework/Vdom";
import {render} from "../framework"

//declare empty list to store data in
let myList = []

//create App
function App() {
    return (
        <div>
            <GeneralInfo list={localStorage}/>
        </div>
    )
}

//render App to the page first time
render(
    <App />,
    document.getElementById('app')
)

function GeneralInfo() {
    //add imput value to list and re-render the page
    function addText(e){
        if (e.key === 'Enter'){
            let textToAdd = e.target.value.trim()
            myList.push(textToAdd)
            console.log("TEXT ON", textToAdd, myList)
            e.target.value = ""
            render(
                <App />,
                document.getElementById('app')
            )
        }
    }
    return (
        <div>
            <input placeholder="SOME TEXT HERE" onKeyUp={addText}></input>
            <ul>
                {myList.length > 0 ? myList.map(el => {return <li>{el}</li>}) : "nothing added yet"}
            </ul>
        </div>

    )
}

console.log("GREAT JOB!!!")
```

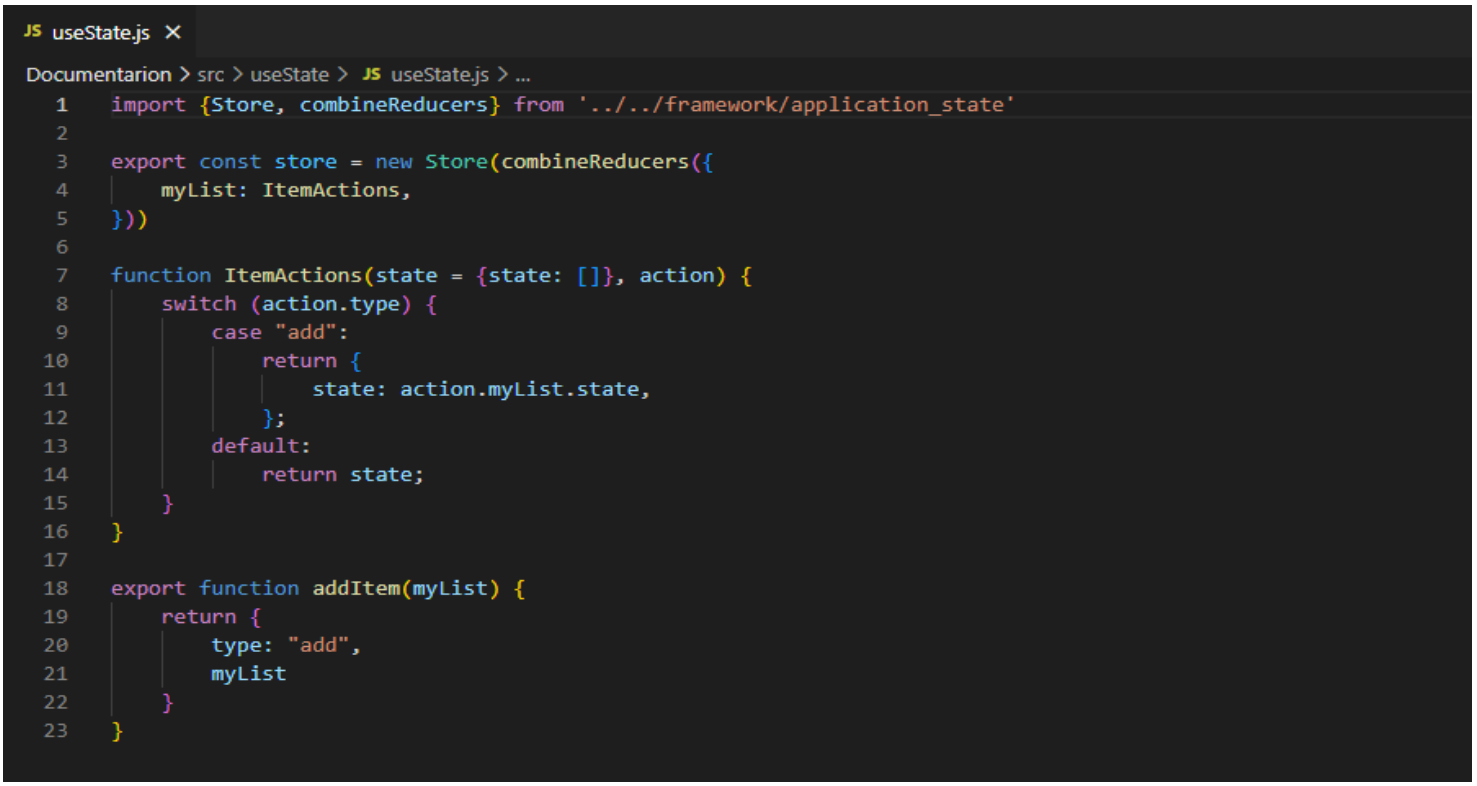


Great! Now you are ready to play with the page and create some more your own elements and render them on the page!

STATE MANAGEMENT

Framework gives you an opportunity to manage your app state as well. To do so you need to create a new variable using Store class from /framework/application_state.
Here you can store your data. To update data automatically use the “combineReducers” function.

Example:



```
JS useState.js ×
Documentarion > src > useState > JS useState.js > ...
1   import {Store, combineReducers} from '../../framework/application_state'
2
3   export const store = new Store(combineReducers({
4       myList: ItemActions,
5   }))
6
7   function ItemActions(state = {state: []}, action) {
8       switch (action.type) {
9           case "add":
10              return {
11                  state: action.myList.state,
12              };
13          default:
14              return state;
15      }
16  }
17
18  export function addItem(myList) {
19      return {
20          type: "add",
21          myList
22      }
23  }
```

now you can use “store” and “addItem” in your code:

```
import VDom from "../framework/Vdom";
import {render} from "../framework"
import {store, addItem} from '../src/useState/useState'


const newItem = (value) => {
    return {
        id: Date.now(),
        text: value
    }
}
//create App
function App() {
    let state = store.getState()
    return (
        <div>
            <h1> Title </h1>
            <GeneralInfo list={state.myList.state}/>
        </div>
    )
}
```

```
function GeneralInfo(fullList) {
    function addText(e){
        if (e.key === 'Enter'){
            let textToAdd = e.target.value.trim()
            if (textToAdd.length > 0){
               let storeList = store.getState().myList ?   store.getState().myList : [];
               storeList.state.push(newItem(textToAdd));
               e.target.value = ""
               store.dispatch(addItem(storeList));
               rerenderPage(store.getState())
            }
        }
    }
    return (
        <div>
            <input placeholder="ENTER SOMETHING" onKeyUp={addText}></input>
            <ul>
                {fullList.list.length > 0 ? fullList.list.map(el => {return <li
key={el.id}>{el.text}</li>}) : "nothing added yet"}
            </ul>
        </div>

    )
}
```

```
function rerenderPage(){
    const state = store.getState()
    render(
        <App state={state}/>,
        document.getElementById('app')
        )
}
```

and to start all project:

```
rerenderPage()
```

ROUTING

In addition to data you can also store routing paths during the usage.
Just use similar logic to state management example:
Update store variable:

EXAMPLE

```
export const store = new Store(combineReducers({
    myList: ItemActions,
    location: locationReducer,
}))
```

define functions: a`la :

```
const SET_LOCATION = 'SET_LOCATION'
const locationInitialState = {current: window.location.pathname}
export function locationReducer(state = locationInitialState, action) {
    switch (action.type) {
        case SET_LOCATION:
            return {
                current: action.location
            }
        default:
            return state
    }
}

export function setLocation(location) {
    return {
        type: SET_LOCATION,
        location
    }
}
```

use createBrowseHistory in index.js:

```
import {store, addItem, setLocation} from '../src/useState/useState'
import {createBrowserHistory, Link} from "../framework/router";

const history = new createBrowserHistory
history.listen(store.dispatch.bind(store), setLocation)
```

add Links from /framework/router to your page and enjoy!

```
function Filters() {
    return (
      <div>
        <li><Link history={history} to='/'> MAIN PAGE</Link></li>
        <li><Link history={history} to='/about'> ABOUT</Link></li>
        <li><Link history={history} to='/contact'> CONTACT</Link></li>
      </div>
    )
}
```

```
//create App
function App() {
    let state = store.getState()
    return (
        <div>
            <h1> Title </h1>
            <GeneralInfo list={state.myList.state}/>
            <Filters />
        </div>
    )
}
```

That`s it!!!