LightJS Documentation

Author: Vasily Sukhov

Overview and installation

55% percent of the world web traffic is made by users on mobile devices.

And phones on average are not very powerful compared to normal computers.

On top of that, front-end Javascript frameworks like : ReactJs , Angular or VueJs , typically use a lot of computing resources to run which is not very ideal for mobile devices. This framework is meant to be a lightweight alternative .

Before you start, you will need to have installed Nodejs

Installation Option 1

Download `template_project/` and start

Installation Option 2

import the `LightJs` folder into your `src` that contains all your scripts

Important note: you are going to need a bundler for your scripts , in our examples we use webpack and webpack-cli

So now run this command in your root directory

```
npm init
```

Then we install webpack and webpack-cli (or the bundler of your choice), so we run this command

```
npm install webpack webpack-cli
```

After that, go to `package.json`

and in the key `"scripts"` we add: `"build" : "webpack --mode production"`

so the `package.json` should look like

```
{
  …
  "scripts": {
    "build": "webpack --mode production"
    //your other commands
  },
  "dependencies": {
```

```
    "webpack": "^5.88.2",
    "webpack-cli": "^5.1.4"
  }…
}
```

Here is how your project should look like:

```
dist/
     index.html
     index.css
     main.js – the bundled version of src/index.js
Node_modules/
  webpack
  webpack-cli
  your_modules…
Scr/
     LightJs/
          Element.js
          LICENSE
          LightJs.js
     index.js
```

here is the a simple example to see if it works

index.js

```
import Lightjs_Node from './LightJs/Element.js'
import Lightjs from './LightJs/LightJs.js'


class App extends Lightjs_Node{
    Init(){

    }
    render(){
        return (`<div>
            <p>Hello world , the script is working</p>

        </div>`)
    }
}

var module = new Lightjs(App)
```

index.html

```html
<!DOCTYPE html>
<html>
    <head>
       <link rel="stylesheet" type="text/css" href="./index.css">
    </head>
    <body>

        <script  src="./main.js "></script>
    </body>
</html>
```

Now run

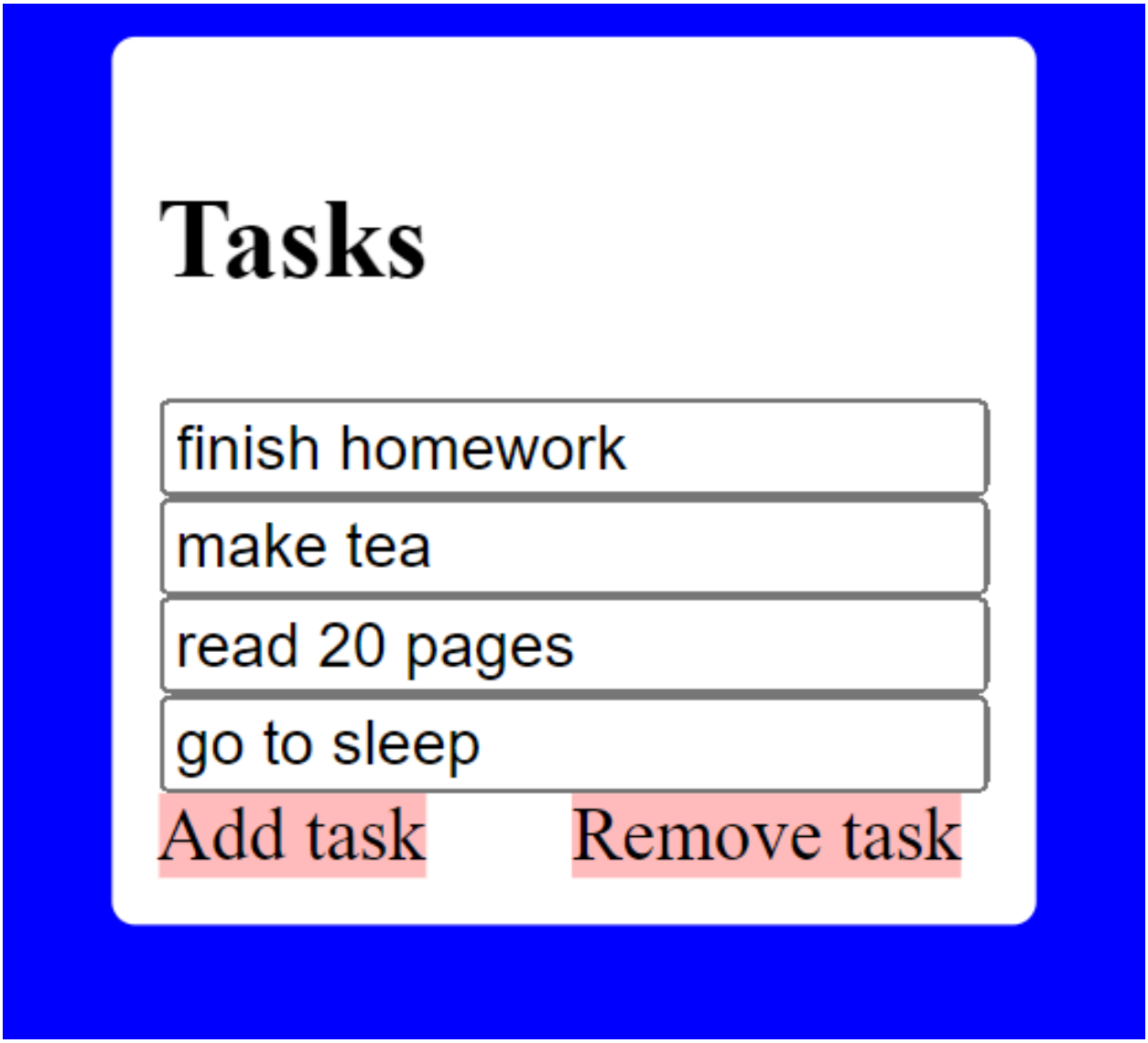
```
npm run build
```

If everything is fine, in the browser when you run index.html you should see the message :

```
Hello world , the script is working
```

Next, let’s see a more complex example

In this example we will make a web page that allows you to make tasks



Index.js

```
import Lightjs_Node from './LightJs/Element.js'
import Lightjs from './LightJs/LightJs.js'


class App extends Lightjs_Node{
    Init(){
        // create data
        this.tasks_data = []
        this.Create_Child(tasks, {tasks : this.tasks_data})
        this.Create_Child(Buttons , {add_button : ()=> {this.add_task()} ,
remove_button : ()=> {this.remove_task()}})
        //create children
```

```
    }
    Render(){
        // render chilren
        this.children[0].Render_Element({tasks : this.tasks_data})
        this.children[1].Render_Element()

        return (`<div class = 'main_container' >
                    <div class = 'display_flex' style = 'width : 100%;'>
                       <h2>Tasks</h2>
                    </div>
                    <div>
                       ${this.children[0].html_result}</div>
                    <div>
                       ${this.children[1].html_result}</div>
                    </div>`)
    }
    add_task(){
       this.tasks_data.push("")
       this.Root.Render()
    }
    remove_task(){
      this.tasks_data.splice(this.tasks_data.length - 1 , 1)
      this.Root.Render()
    }
}









class tasks extends Lightjs_Node{
    Init(){

    }
    Render(props){
        let tasks = props.tasks
        let html_task_data = ""
        tasks.forEach(element => {
            html_task_data += `<div class = 'display_flex'><input value =
'${element}'></input></div>`
        });
        return `<div class = 'display_block'>${html_task_data}</div>`
```

```
    }
}
```

```
class Buttons extends Lightjs Node{
    Init(props){

       this.add_task_callback = props.add_button
       this.add_button_id = "add_button"

       this.remove_task_callback = props.remove_button
       this.remove_btn_id = "remove_button"

    }
    Render(){

       this.Root.Add_Effect(()=>
{document.getElementById(this.add_button_id).addEventListener("click" ,
()=>{this.add_task_callback()})})

       this.Root.Add_Effect(()=>
{document.getElementById(this.remove_btn_id).addEventListener("click" ,
()=>{this.remove_task_callback()})})

       return (`<div class = 'display_flex'>
                     <div class = 'display_flex' style = 'width : 50%;'>

                        <div class = 'btn' id ='${this.add_button_id}'>Add
task</div>

                    </div><div class = 'display_flex' style = 'width : 50%;'>

                        <div class = 'btn' id = '${this.remove_btn_id}'>Remove
task</div>

                    </div>
                </div>`)
    }
}


var module = new Lightjs(App)
```

index.css

```css
body{
    background-color: blue;
    display: flex;
    justify-content: center;
    align-items: center;
}
.btn{
    background-color: #FFBBBB;

}
.main_container{
    background-color: white;
    padding: 10px;
    border-radius: 5px;
}
.display_flex{
    display: flex;
}
.display_block{
    display: block;
}
```

So in this case we have 3 nodes

App

tasks

Buttons

In the App class the first thing we do is create a variable that holds

a list of task which we call tasks_data

then we have 2 functions to add or remove tasks

add_task()

remove_task()

Note : both these functions have the line:

this.Root.Render() – this command rerender's the app after the values have been updated

After these functions we create 2 child nodes :

tasks

Buttons

Note: we pass the 2 previous functions as callbacks to the Buttons node

and pass tasks_data to tasks

App.Render first render its child nodes

Then take their results and return its own result

*Note: App.Render only passes tasks_data to the tasks node

*It does not need to send the callbacks to the Buttons node because it stored them

Now we will look at the class tasks

In its Init() it does nothing

In its render function, it creates a html input object for every task then wraps them inside a div and return's it

And finally the Buttons node:

Buttons.Init(props) first stores the callbacks it received then creates 2 constant variables which represent the id of the 2 button it will create : add task and remove task

In Buttons.Render() we can see the command : this.Root.Add_Effect(…..

The function this.Root.Add(callback) runs the callback we gave it after the page has rendered

In our case we use it to add event listeners to buttons

After these effects , the function returns the html data of the buttons

Now if you run

```
npm run build
```

you should see the result you saw in the previous photo

So now we will dive deeper into what happens when we run index.js

The first thing we do is create a class app that extends from LightJs_node

As the name suggests the class acts as a node , these nodes are the building blocks for our app which is a node tree

The constructor takes in 3 arguments : address , Root , props

Then the object stores :

`this.Root` – the root object of the framework its type is: LightJS
`this.Address` – the address of the node, type : str

`this.Name` – this a variable that can be change in Init() : type : str

`this.children` – array that holds its children of type : Lightjs_Node

`this.html_result` – str that holds html data from the render: type : str

`Methods:`

`Init(){……}` is called when the object is constructed
used to create children and states

`Create_Child(child_class ,props )` this method create's a child of <child_class> and appends it to this.children

`props` is argument that is a dictionary that is used to pass data to the child

`Render_Element(props)` called by the parent , this method calls this.Render(props) and stores the result in this.html_result

`Render()` this returns str data of the node render

Example of use case

```
class App extends Lightjs_Node{
    Init(){
        //create children
        // create data
    }
    Render(){
        // render chilren
        return (`<div> ${this.children[0]}</div>`)
    }
}
```

You can also run code after the page has been rendered with `Root.Add_Effect(callback_)`

This is used to add event listeners to the page and make it interactive

Now we will look at the LightJs class.
LightJs is the framework's renderer

The constructor ask for the a node class it will create under the first node.

The constructor always creates a first node under itself.

```
this.virtualDom = new Lightjs_Virtual_DOM("",this,{_app_class : app_class})
```

it creates a node called `virtualDom` and passes itself as the `root` and the `app_class` as a prop , then `virtualDom` creates `app_class` under itself.

this is the render function of the root

```
Render(){
        this.virtualDom.Render_Element()
        document.body.innerHTML = this.virtualDom.html_result
        this.run_effects()
    }
```

So it calls `render_element()` of `virtualDom` which will call `render_element()` of your App.

Here is a list of methods of the root object `LightJs`

- `this.Render()` – renders the page

- `this.Add_Effect(function_callback)` – run's a callback post render ex: () => {console.log("hello world")}