Most common design patterns for Front-End with JavaScript (Real-world examples)

#beginners #javascript #webdev #react

Luis Castillo Jun 29 • 3 min read

Hello everyone, in this post, I want to show you how can easily we can implement some common design patterns in your projects. These patterns help us to have a maintainable, flexible, and readable code. You will see the advantages when you need to add more functionalities without do a lot of modifications in the code.

So now, let's code!! 💻

1.- Module pattern.

The module pattern is one of the most common patterns in the JavaScript world and it's it is very useful to encapsulate the logic of functions and variables in the same scope and manage the access of them, something similar to access modifiers (public, private, etc).

There are a lot of examples on the internet with different variations but I tried to create an example as simple it is possible.

Note: We can see this pattern in **ES6** with *import/export* syntax.

Complexity: ⚡

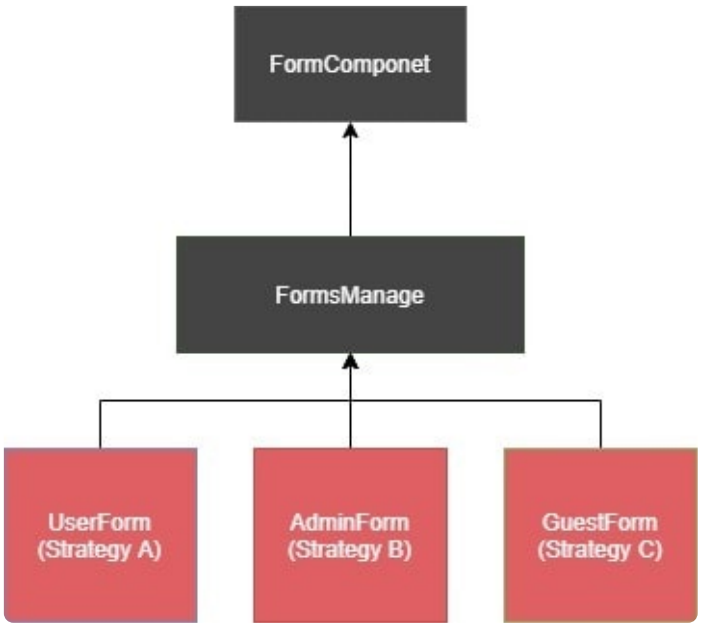
```
var module = (function () {
  let options = {color:"red"}
  /*
    private code here
```

```
    options["size"] = 12;
  };


  /*
    public code here
  */
  return {
    getOptions: function () {
      setSize();
      return options;
    }
  };
})();
module.getOptions();
```

2.- Strategy pattern.

The Strategy design pattern is very used when we have similar tasks and we need to change between tasks in the runtime.
This pattern helps us to eliminate a lot of if-else sentences, to do that simply we have to encapsulate the tasks in small chunks and use an object literal to access our concrete strategy.



This pattern is very simple to implement in javascript because you don't need interfaces or any hard implementation.
Use case: Imagine that we have a dropdown with different user types (normal user, admin, and guest), and we want to display a form on the same page dependence on which user type was selected.

Here is an example with *React*, but you can apply in other JS frameworks.

Complexity: ⚡⚡

```
// React components section
import React from "react";
import UserForm  from "./userForm";
```

```
/*
* This object literal will help to encapsulate all the forms that could we have.
*/
const FormsManage = {
  user : {
    render(props){
      return <UserForm {...props} />
    }
  },
  admin:{
    render(props){
      return <AdminForm {...props} />
    }
  },
  guest:{
    render(props) {
      return <GuestForm {...props}/>
    }
  }
};

/*
* Main form component
*/
const Form = (props) => {
  // here we are getting the form by type
  const userForm = FormsManage[props.type];
  return userForm.render(props);
};
export default Form;
```

3.- Builder pattern.

The builder pattern is used when we need to create complex objects with different variations and also we want to have the flexibility to modify the construction process without impact the object representation in self.
I tried to create an example that we can use in the *real-world*.

Use case: How many times we need to transform the API data into a format that our *third-party* component understands, for this case, we can use the builder pattern to create the object that the component needs, and also separate the construction logic.

Complexity: ⚡⚡⚡

```
/*
* Mock class
*/
class DataTable{
  constructor(data ,options){
    this.data = data;
```

```
  getData(){
    return this.data;
  }
}

/*
* Builder class to create DataTable objects.
*/
function DataTableBuilder () {
  let defualtOptions ={ width:100, height:200, headerFixed: false };

  /*
  * Method to make the format required.
  */
  function generateFormattedData(data,header){
    return data.map(item => {
      let result = {};
      item.forEach((val,idx) => {
          result[header[idx] || "df"+idx] = val;
      });
      return result;
    });
  };

  /*
  * Public steps methods
  */
  return {
    addHeader(header){
      this.header = header || [];
      return this;
    },
    addData(data){
      this.data = data || [];
      return this;
    },
    addOptions(options){
      this.options = { ...defualtOptions, ...options};
      return this;
    },
    build(){
      const formattedData = generateFormattedData(this.data,this.header);
      return new DataTable(formattedData,this.options);
    }
  }
};

/*
* Data needed to build the Datatable object
*/
const header=["name","age","position"];
const rows = [["Luis",19,"Dev"],["Bob",23,"HR"], ["Michel",25,"Accounter"]];
const options = { headerFixed:true };
```

```
*/
const dt = new DataTableBuilder()

                .addHeader(header)
                .addData(rows)
                .addOptions(options)
                .build();
dt.getData();
```

Conclusion.

In the software development world exists many design patterns and all of them have their qualities, but it is work of us as Developers, understand and analyze which of them add real value to our project and not more problems or complexity.

If you have some design patterns that have been useful for you, please share in the discussion section or if you want to implement one of the previous ones and you need a hand, let me know and I can help you. 😉

Discussion

Subscribe

DEV

Add to the discussion

Luis Castillo

JavaScript lover, Thinker and Meditation Fan

Follow

WORK
Full Stack Developer

LOCATION
Mexico

JOINED
Apr 28, 2020

More from Luis Castillo

3 Courses to Become a Better Software Developer 2020

What are the needed qualities to be a tech-lead?

#discuss #career #leadership #webdev

Top VSCode Extensions to be a happier FrontEnd.

#productivity #javascript #beginners #webdev