Software Development

Modularizing JavaScript code to make it more stable and readable

March 5, 2019

Johnny Dunn
Full Stack Engineer

Do you seem to be repeating logic, or writing the same structure of if-else statements repeatedly?

This usually means you can abstractify and encapsulate the system to make it more modular—can you wrap your code into functions, and keep those functions contained within classes? If you're programming in an object-oriented language / paradigm, generally, modularizing code is one of the most useful things you can do.

Let's take a real-world example of a web application that sends a request to a stock market website to parse and display prices. What are the the functions, or parts of the program, that need to happen when a user interacts with the application?

First, a HTTP request is sent by the app's backend to fetch the data to display, in this case, being HTML contents to parse (tThis is assuming that the stock market

site has no API available returning structured data). You might start off writing the logic for this interaction by hardcoding the requisite URL in the same line of code containing the request, which is straightforward and successfully returns what you want.

Next, you must present the returned HTML in a readable way for the user, and one that works with our UI. So you add multiple lines of code performing string replacements by looking for regexes representing non-ASCII characters, HTML tags, and line breaks. When all the regex replacements are done, you're left with the pure textual information found on that page, which, in this hypothetical app, will be perfectly clean and contain only information relevant to today's stock prices.

Here's what that would look like in JavaScript:

```
// This is just example code; copying and pasting
// into your browser console probably won't work
// because of CORS
let data;
fetch('https://www.realstockmarketsite.com')
.then((res) => {
   console.log(res.text());
   return res.text();
})
.then((html) => {
   data = html;
});
// Strips out "<'text'>" which includes HTML
// https://www.regextester.com/93515
data = data.replace(/<[^>]*>/g, "");
// Removes non-ascii chars
//https://stackoverflow.com/questions/20856197/remove-non-
data = data.replace(/[^\x00-\x7F]/g, "");
// Removes newlines
```

```
data = data.replace(/\n/g, "");
// Assume we can get stock information in a meaningful
// way by just removing non-valid chars (probably not
// realistic in real-world sites)
document.getElementById("stocksData").innerHTML = data;
```

Now you have a basic web application that fetches stock data and displays it to the user on load. Let's add some more complexity now with a few new feature: allowing the user to see stock data related specifically to a company, by using a ticker symbol / ID appended to the requested URL as a parameter.

If you were following along with all that, then you should immediately recognize what we're building is a typical web crawler. If you were thinking about object-oriented principles in mind, and wrote a function to make the web requests, it would be as simple as connecting that function to an input form within the UI, and allowing users to enter in a ticker ID to get a specific listing from the stock site. You could write another function that accepts an extra

parameter for the ID, or you could modify the original requests function and make the ID parameter optional and defaulted to an empty string. Now you've avoided any duplication of logic in your code, and have maximized readability and brevity, both of which are generally desirable results when refactoring.

But you might have noticed that we are now left with one function, and multiple lines of logic outside that function for cleaning up the text, and that is actually where we will see the most benefits in refactoring / modularizing the code, even though it's not necessary to encapsulate the regex replacements in a function now as of now (since it's only done once, on the stocks data before displaying it to the user).

```
let data;
let tickerId = userInputValFromForm['data'];
data = fetchStocks('https://www.realstockmarketsite.com',
function fetchStocks(url, tickerId="") {
    if (tickerId !== "") {
        url = url + '/' + tickerId;
    }
    fetch('https://www.realstockmarketsite.com')
    .then((res) => {
        return res.text();
    })
    .then((html) => {
        return html;
    });
}

data = data.replace(/<[^>]*>/g, "");
data = data.replace(/[^\x00-\x7F]/g, "");
data = data.replace(/\n/g, "");

document.getElementById("stocksData").innerHTML = data;
```

If we do another refactor and encapsulate the string replacements into a function, we'll get something like this:

```
function cleanTextData(data) {
   let cleaned = data;
   cleaned = cleaned.replace(/<[^>]*>/g, "");
   cleaned = cleaned.replace(/[^\x00-\x7F]/g, "");
   cleaned = cleaned.replace(/\n/g, "");
   return cleaned;

data = cleanTextData(data);
document.getElementById("stocksData").innerHTML = data;
```

Or better yet, an even more modular implementation, which has more lines of code, but will afford us more flexibility and extensibility in the future:

```
function stripHtmlTags(text) {
   return text.replace(/<[^>]*>/g, "");
}

function stripNonAsciiChars(text) {
   return text.replace(/[^\x00-\x7F]/g, "");
}

function stripNewlines(text) {
   return text.replace(/\n/g, "");
}

function cleanTextData(data) {
   let cleaned = data;
   cleaned = stripHtmlTags(cleaned);
   cleaned = stripNonAsciiChars(cleaned);
   cleaned = stripNewlines(cleaned);
   return cleaned;
```

```
}

data = cleanTextData(data);
document.getElementById("stocksData").innerHTML = data;
```

Besides making the code more descriptive and organizing it into contained sections, we also gain lots of practical benefits that can be used to improve the software and make it more robust. We have a var $tickerId which is obtained from an input form, which actually, if you've any experience dealing with user input, could not be left in its raw form in production environments. The program could be attacked via SQL injections, cross-site exploits (XSS), and more. Which gives us a perfect and natural use case of our newly created cleanTextData() function, doesn't it?

Let's take a look at our final code, after all the refactoring we've done:

```
let data;
let tickerId = userInputValFromForm['data'];
tickerId = cleanTextData(tickerId);
data = fetchStocks('https://www.realstockmarketsite.com',
data = cleanTextData(data);
document.getElementById("stocksData").innerHTML = data;

function fetchStocks(url, tickerId="") {
   if (tickerId !== "") {
       url = url + '/' + tickerId;
   }
   fetch('https://www.realstockmarketsite.com')
   .then((res) => {
       return res.text();
   })
   .then((html) => {
       return html;
   });
```

```
}

function stripHtmlTags(text) {
   return text.replace(/<[^>]*>/g, "");
}

function stripNonAsciiChars(text) {
   return text.replace(/[^\x00-\x7F]/g, "");
}

function stripNewlines(text) {
   return text.replace(/\n/g, "");
}

function cleanTextData(data) {
   let cleaned = data;
   cleaned = stripHtmlTags(cleaned);
   cleaned = stripNonAsciiChars(cleaned);
   cleaned = stripNewlines(cleaned);
   return cleaned;
}
```

Modularizing code not only makes it tighter and more readable, but also allows new systems to be built on top of existing features, while maintaining object-oriented design principles of low coupling and high cohesion. We could take this one step further and encapsulate the text replacement functions within their own class / module, that could be integrated with any other parts of our software where necessary.

0 Comments Clubhouse Login

Recommend Tweet Share Sort by Best

Start the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

Be the first to comment.

Subscribe Add Disqus to your siteAdd DisqusAdd Disqus' Privacy PolicyPrivacy PolicyPrivacy

Product

Features
Pricing
Enterprise
Integrations
Write beta

Company

About Us
Customers
Careers
Press
Contact
Brand Guidelines
FAQ

Developers

REST API Docs
Webhook API Docs
Community
Open Source Projects
Developer How-Tos
Write for Clubhouse

Resources

Help Center
Blog
Webinars
Status
Referral Program
Release Notes

The Clubhouse Blog

Collaborate faster at scale with Group @mentions and notifications

Execute commands more efficiently with the Action Bar