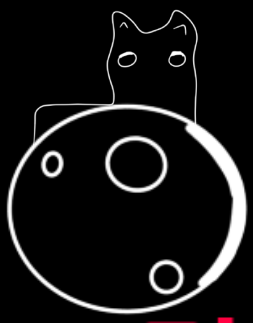


SkyPenguinLabs

SkyPenguinLabs PRGC3

The Art of Frontend Validation for UX

SPL-PRGC3

Introduction

> The Art of Frontend Validation for UX

When developing frontend applications, especially using frameworks such as React, Vue, or AngularJS, there are many times varying detection methods may be implemented into the apps logic to pass information to other components, such as a login request builder, or server-side connection module.

However, the issue with client-side apps is that if a user is stupid enough, or if the user is intentionally trying to ruin the app, it can be very easy to break components in the frontend application by manipulating the data on the client.

Let me throw a hypothetical at you - A web application built using ReactJS contains a tab for configuring user settings. One of the settings allows users to switch the email address the account is under. Logically speaking, this sets a users email (*before sending it to the server*) in the browser's `localstorage` when the user hits a button that says 'save email change. When the user hits *'save all settings,'* the user takes all of the settings applied to `localstorage` to the server. However, if the user chooses to change the data in the `localstorage` to some arbitrary, unknown value, and it hits the request module, how will the module handle that?

Now, ideally, server-side validation in this hypothetical scenario would prevent anything malicious from happening if it were done right. However, we want to take a different approach to the way things get handled in apps. Instead of a security focus, we want to make sure that even small user changes like this will prevent unnecessary processing on the server side. Saving a VPS bill a few cents in processing power.

Consider you are in a seriously restrictive environment, and have a large number of stages for validating and parsing/deserializing data structures, then it's crucial to ensure that stuff is handled as much as possible on the client side before sending to the server to go through all of these steps or even attempt to start the process for data that is certain to fail.

Now, of course, this is NOT saying that it's in any shape or form a security measure because **its not**, but in some cases, had the server validated the data in `localstorage` before sending it to the server, there would have been many

chances the server could save a few hundred requests in a production environment, even local, when there was a guarantee for the data to fail. Even in large applications, such as enterprise platforms with THOUSANDS of users, how many times would a user pressing a button to 'save settings' in turn make the server return an error to the client saying 'settings already saved', that the client could have caught on the app before sending it to the server?

Some might misinterpret this as advocating for frontend validation as a security measure, but that's not the case. The focus here is on improving user experience and using frontend validation as a chance to prevent resources from being used too frequently on your apps by users without the intention. Frontend validation helps catch issues early, preventing users from encountering vague or unnecessary server-side errors. This is one of our many goals.

Powered by SkyPenguinLabs

Table of Contents (ToC)

> The Art of Frontend Validation for UX

This lesson is a much more lightweight lesson, as it is theoretical and contains only 3 primary sections.

- **Section 0x00 |** Prerequisites - *(What you may need to know before reading)*
- **Section 0x01** | A Misconstrued Purpose - *(Kicking this off by getting people to understand what we mean when we talk about frontend validation which also talks about how it gets engineered into the design of the application)*
- **Section 0x02** | Real World Examples- *(Real world examples using ElectronJS)*
- **Section 0x03** | Conclusion - (*Concluding this entire lesson and what will be expanded on in the meer future*)

Powered by SkyPenguinLabs

Section 0x00

> Prerequisites

This lesson aims to be fit for anybody with a basic understanding in development both frontend and backend. This means having understood the basic proponents that go behind RESTful APIs, Client Side Apps versus Static Sites, and have a basic understanding of how server sides work.

This lesson does not cover any code or have any environments that are being used.

This lesson isn't meant to be overly complex or require deep technical mastery to grasp. The goal is to break down a fundamental concept—how the lack of a clear connection between frontend validation and backend processes can lead to security vulnerabilities, while also showing how thoughtful frontend validation can significantly improve the user experience.

- *The content provided here is intended for educational and conceptual purposes only. It should **not** be interpreted or relied upon as formal security advice or some over complex technical jargon. Any misinterpretation of this information as actual security guidance is solely the responsibility of the reader. While notes contain information about security concerning the frontend, the explanations are mere samples of how the frontend works in conjunction with backend validation.*

That being said, enjoy the lesson! The first section begins on the next page.

Section 0x01

> A Misconstrued Purpose

One of the things I want to start this lesson out with is to express something I have observed during the time I spent on development teams. This is the misconstrued purpose behind what frontend validation is.

First off, frontend validation, directly is the concept of building out logical steps within a client-side application built out using frameworks such as ReactJS and VueJS directly for data input/output validation.

The primary purpose behind frontend validation is to ensure that before sending anything to a server, or remote service, or even, another component within the client side application (such as a login panel, or a user dashboard component) the data being utilized by the component or service will not revert when attempting to load.

- **Important Note:** For those who don't know, when frontend apps are programmed, components are often initialized with data, or initialized expecting specific data to exist within the application. If the data does not exist, meaning a request may have failed to fetch the data (in a hypothetical scenario), the component may break, look wonky and produce errors in the console that seem related to the code but aren't, its just because the app could not load data and faulted at a specific point. Modern compilers for building web applications, such as the TypeScript compiler is good enough at ensuring developers make data certain before loading components, but there are still ways of bypassing this, and with coe AI slop becoming more and more common, real frontend validation is slowly becoming less prominent in everyday applications, as its often ignored, or in some cases overwritten with tests code.

Unfortunately, as I mentioned in the introduction and notes, some developers do mistake this as an actual security measure for server-side components or the security behind the client-side loading of sensitive procedures that rely on information provided by an external service, system, or user.

The problem is that while frontend validation can in some shape or form *__protect__* your server, it is not really *__securing__* it. And I think this is something a lot of people forget.

Putting this into security theory is simple:

- Protection is passive and helpful, but not authoritative. Security is active, enforced, and cannot rely on user-controlled environments.

In this case, frontend validation can be used to protect the server from bad, malformed requests made by users by accident.

- Note: Some people would make the argument 'well you can't stop a user from using curl'. Yes, we understand that, there are technically ways to block curl specifically, but it's rarely worth it to try. This is not our goal. Any user who intentionally uses curl to fetch a web apps page or information is at that point no longer the user groups we are trying to handle apart of that application __UNLESS__ it was designed or engineered as a feature into the software, at which point, its context dependent on the scenario.

Understanding the difference is HUGE here. Because what we will be talking about today is very specific scenarios, more-so production scenarios which involve needing to optimize web applications with frontend validation techniques to prevent unnecessary server load.

The idea of this is that even if a user changes items in the scenario we shared in the introduction to this lesson, it will still revert before being sent to the server unless the client purposefully removes that portion of the component.

Now, the important thing is to also talk about how validation on the frontend can be fit specifically around the client-side engineering.

The section on the next page covers this more in depth.

Engineering Frontend Validation into Design

When designing a web application, specifically one that relies on a lot of user input, keys, and various forms of data from external environments, we need to ensure that everything gets validated.

So how exactly is the validation ideas on the frontend thought out?

Well, the same way they are on the server side.

After the design of the web application is thought out, and a baseline between all of the components and the information they need has been solidified, a good engineer would take each of these components and analyze

- What data it needs
- Where that data comes from
- How it uses the data

Let's pick an example component in an application.

The most basic - A simple user login component which takes an input email address, password, and phone number. The login component, when called to submit the form, passes it to a remote request module in the client-side app so it can send the data to the designated server.

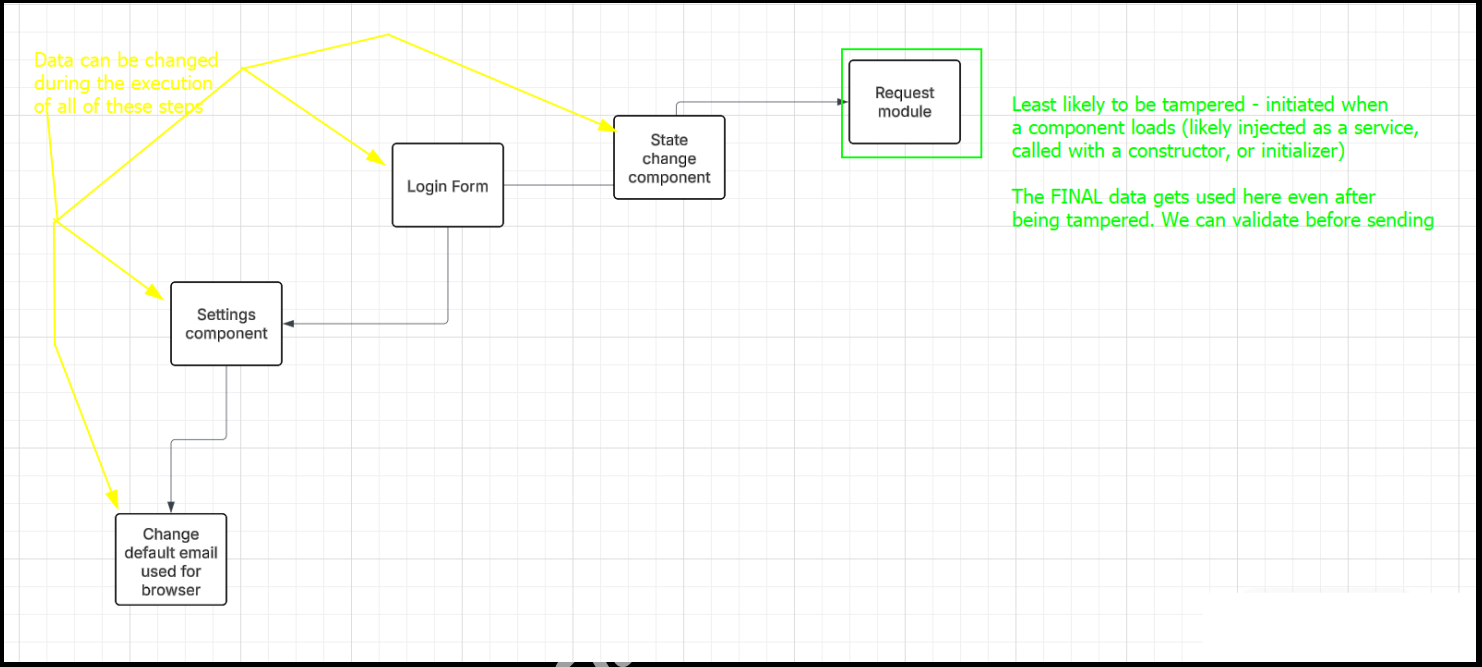
This component

- **NEEDS** - a valid email, password, and phone number
- **WHICH COMES FROM** - The user
- **AND IS USED** - by the request module to sign the user into the app via remote server authentication (*not authorization in this scenario*)

At this stage, the focus shifts to identifying the final point where the data enters the system, the exact landing point where the user input reaches the application logic or server for processing.

We care about the final point the data is LAST referenced during the execution to send it to the server because this is the point where it is least likely to be modified or accidentally changed by a user, or even a conflicting bug in the application (*depending on where the bug is in the component*) after it gets validated.

Let's visualize this.



Request modules when sending data logically send finalized data to the server. So we want to intercept the components after they process, modify, or save the data and call the request module to send it.

Validating anywhere in the yellow will most likely result in a failure, because even if you put validation in the state change, there could still be enough time for data to be changed during the process between the actual use of the request module. This, of course, gets gauged based on the situation.

When this gets visualized, it becomes apparent where somebody who gets paid to look over the app may see this layout and know exactly where to place the validation.

Now, we need to guage its use.

Properly Gauging Frontend Validation

A majority of developers often implement either too many or too little frontend validation, and when they do implement that perfect in-between amount, the moment usually is not the time to work with it.

What do I mean by this?

It comes down to understanding the purpose of frontend validation.

If your application relies on high-performance, responsive, and resource-efficient servers, especially under heavy load or high throughput, then it's worth investing in robust, well-engineered client-side validation. Filtering and verifying data before it hits the server can reduce unnecessary processing, save bandwidth, and help avoid avoidable server-side parsing or rejection logic.

On the other hand, for simpler use cases, like a basic contact form or business info submission box, lightweight validation is usually sufficient. You're not optimizing for scale or performance in these scenarios; you're just ensuring the user enters reasonable data and gets immediate feedback.

Additionally, it's also important to gauge the information you are giving to the client to implement those validation mechanisms.

Let's consider a scenario where a web application is designed to run a specific cryptographic operation on some data the user inputs to validate if it's safe or not to enter remote storage. If the operation is proprietary, implementing a validation on the client side by forcing the client to perform this operation gives absolutely anybody with half of 6 billion braincells the ability to ruin your entire server-side security.

This, of course, in production would ideally be caught by security engineers and security testers; however, I must reference this concept because I have seen it many times at least thrown out on the thought table, and I know I would not be the first to say this.

Optimizing Frontend Validation - Baseline Concepts

When implementing frontend validation techniques, like making sure you are not overly beefing up with 20 different techniques for validating the same data, you also need to make sure the components you implement ensure they are optimized for the scenarios you are in.

Especially because validation logic can be heavy, it's important to optimize it. This means focusing on validating only what's necessary on the frontend to avoid unnecessary processing and slowdowns for large or complex input/data forms.

Consider also making the application much more asynchronous in its validation stages, as the UI needs to make sure no components that rely on other processes to finish are waiting or being prevented from executing.

Conversely, implementing debouncing or throttling mechanisms to limit the frequency of validation checks during continuous user input (*e.g., typing in a search bar*) reduces processing overhead. I have seen this implemented in WAY too many apps, and it comes from a design team that does not account for the UX properly. Sometimes, when making requests to a server or remote service, such as Google's address API, users may unintentionally rate limit themselves. This happens because the app sends a request with every keypress. It's seriously annoying and a large waste of memory.

If API rate limits at 6000 Queries Per Minute (QPM), considering a US address on average may reach 38 characters `(Sample: 123 Main Street, Springfield, IL 62704)`

This means that every single time you typed a character and submitted a request to the API, you would end up with 38 queries per address on average. This is a huge performance issue. And as you can imagine, if the data is invalid, and the user backtypes, its even more load for no reason.

While the API is a small example, this is perfect for situations where custom APIs are implemented, and how basic frontend controls can also ensure that the data before being submitted is complete as expected (*e.g, by emitting an event when it's triggered by a user, such as a button press to submit*)

As you can see, all three of these concepts break down primarily into

- Optimizing the validation logic
- Making the logic asynchronous
- And designing proper debouncing techniques

These concepts are what will be the baseline for a large amount of your optimizations, which allow you to properly integrate forms of frontend data validation without making your application bloaty or slow because of the logic.

Once frontend validation gets implemented into the design of the application, then is technically implemented and optimized, the next step is to make sure that all areas of the application are covered, and test cases for the data are properly error-handled if they revert/fail testing.

Let's reach the end of this lesson (next page) so we can move on to the practical samples! O_O

Conclusion

With that all being said, you should be able to comprehend that frontend validation is simply a technique to make the user experience much smoother and a lot more functional on the application. Additionally, we aimed to describe how much of an impact it can have as far as security. This means

- Ensuring the logic on the client does not expose proprietary components or data structures (*if so, server-side it*)
- Ensuring the logic for the validation on the client does not slow down any other major components, which involves optimizing the logic
- Ensuring the logic is well designed and implemented into the proper components/stages of data use in the client

Now that we have this understanding, we can move more into the practical details. In the next section, we will be working with an existing ElectronJS environment to draw out both good and bad examples of frontend validation stemming from the theory discussed throughout this section.

The next section, while defining our code in ElectronJS, will not be over complex, or share any functionality that will spin your head in circles, but it may require some understanding of logical programming.

Either way, I try to break it down ¯_(ツ)_/¯

- **Note**: I used the same ElectronJS app from the lesson I did on building tools using Golang to automate the process of unpacking and scanning compiled ASAR files built by Electron.

Powered by SkyPenguinLabs

Section 0x02

> Real World Examples

Before we dump out the code, I want to make sure we solidify what we are going to be covering. That is shown below.

Code Sample	Description
CS-1 (BAD!)	Frontend validation leaks proprietary cryptographic password routine
CS-2 (OK)	Frontend validation of a phone number in `localstorage` before using it in a POST request
CS-3 (BAD!)	Contains a load of AI generated LLM junk that was designed to validate user inputs, but instead, validates all user inputs every single keystroke which is BAD by design

I will be breaking all of these down in their own section below, but first, lets cover something about these apps.

These apps are in no shape or form intended to be Capture The Flag challenges, if you want to rip them apart, and see how the code works yourself go ahead if you have the skill or know how. But the purpose of these apps is to __demonstrate__ scenarios that often cause user experience issues or are just bad in practice to solidify some of the theory we covered here.

Do not take these as core foundations to build your knowledge, only use them as representations of the knowledge and theory picked up in this lesson.

As you can probably infer, there are 2 bad examples of how frontend validation should NOT be done and one which is actually pretty spot on for the scope. Note that all of these are 3 different scenarios.

- Haxxer, But why the AI generated scenario? Truly, if you are not new to the field of tech right now, or overall development, then you know the current landscape if flooded with vibe coding. Vibe coding is a practice that revolves around using AI modals to generate code for you. In many frontend applications, sloppy validation techniques are created or programmed into applications, and left abandoned or uncared for, because of how little people tend to care about the frontend validating data. You will see AI generated code a lot moving forward, additionally, you may even see over engineering and over complexification in components like validation components. This eventually leads to massive performance issues. I wanted to cover this because I know many people right now that do this day-to-day and care very little about the code the model developed (*on both the backend AND frontend side*).

The scenarios are explained in their own sections below.

Scenario 1 - BAD Frontend Validation for User/Password Form

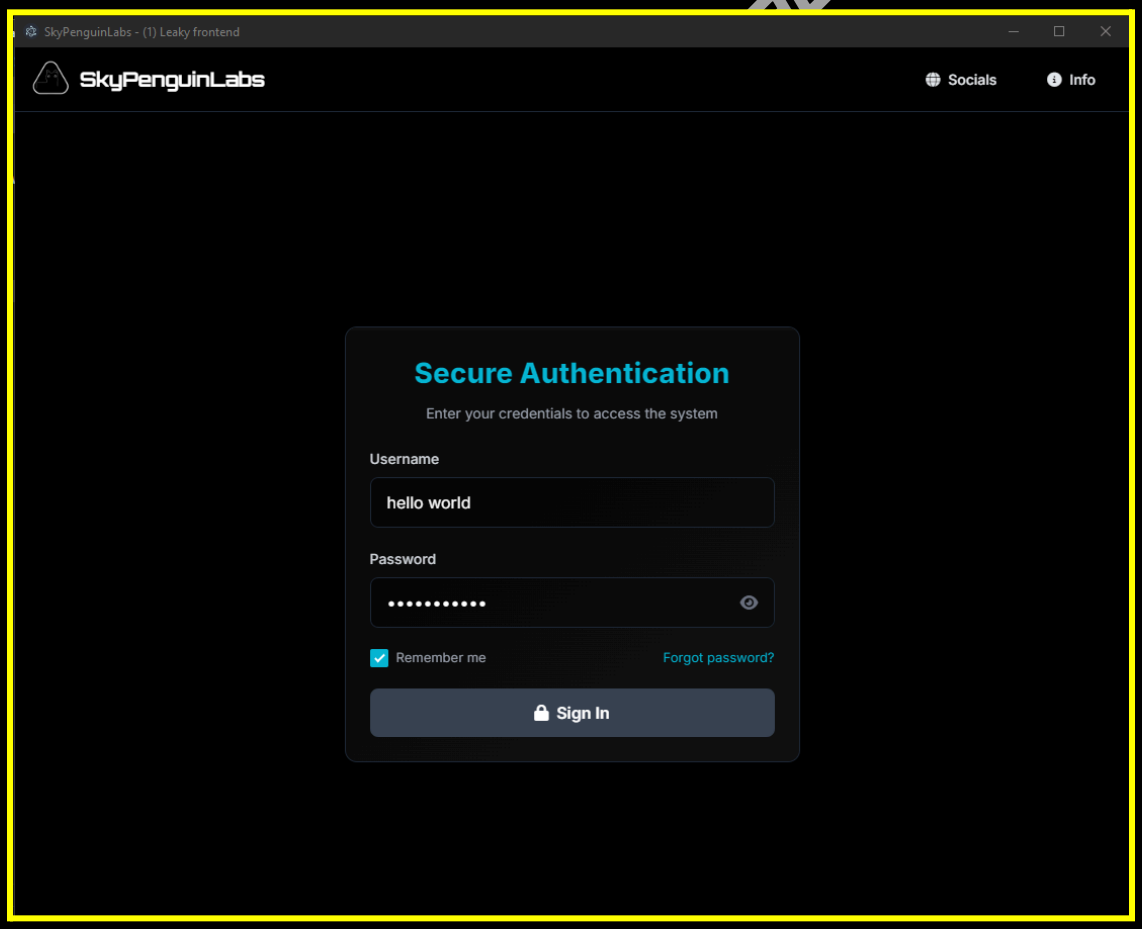
In this scenario, it is described that the application in question (*here*), is designed to take an input username and password, and validate it before sending it to a server. The purpose in the validation is to make sure the username and password can be used to create one-way hashes. However, this application has code inside of it which leaks the secret-key used to influence each cycle each character in the data goes through. The reason this was done is because a developer told an AI model to have a frontend application make sure the username and password are fit to be stored in the database using the proprietary format, to which, the only proper way of handling it in the frontend was to- well, expose it.

This is of course, a very beginner, ammature mistake that in most professional environments would most likely be caught, or, you'd hope. But, this kind of stuff does not just happen with secret keys or secret cryptographic routines. This type of flaw in an applications implementation can result in almost any specific internal-required *(but hidden)* data format/structure/requirement to be exposed.

Check out the application hosted here, shown below, to grasp what we are saying!

- Note: While I will explore the source code of the application, I will not explain the exact block-to-block build of the entire application, because these are demo, compiled, ElectronJS apps, which I am referencing source code too. If you know how ASAR works, have fun unpacking! Since the point of focus is bad and good examples of specifically the validation, going over each app's individual segment of code would end up beefing this article up, and forcing us to raise the price for content that makes no sense- so, I tend not to do that here.

- **- 1) The Application: First lets take a look at the applications frontend.**



As you can see, it's a basic username and input login form. Additionally you may have seen that there was a devtools panel that popped up on render. This was designed for the sake of demonstration and for you to walk through the app.

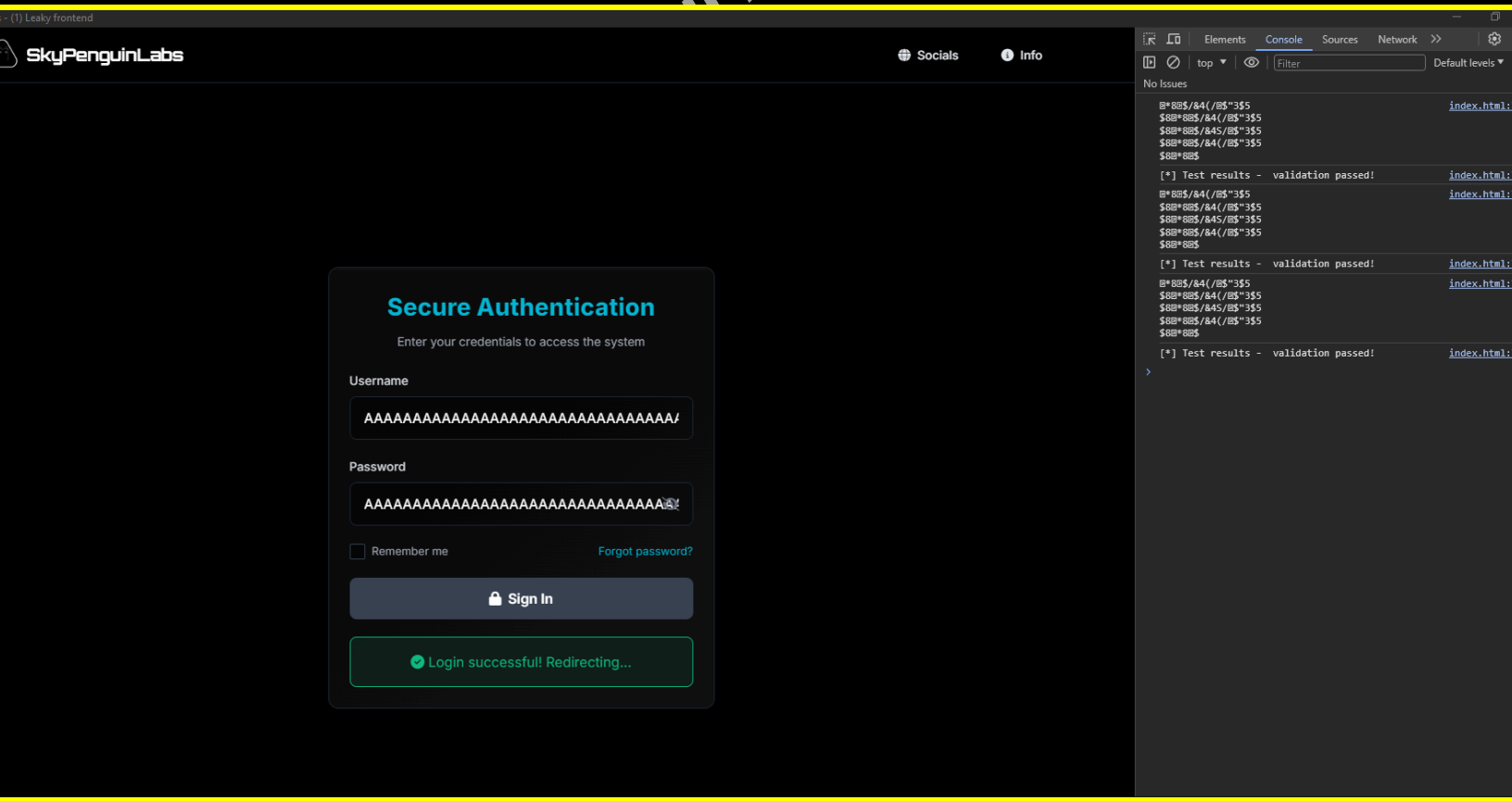
But before you get hungry, let's go ahead and check out the application by running some functionalities.

- **- 2) Giving it some inputs: Lets see how the application reacts to input**

Hitting Enter with 1 input of some numbers as the password and some letters as the username. A generic form input.

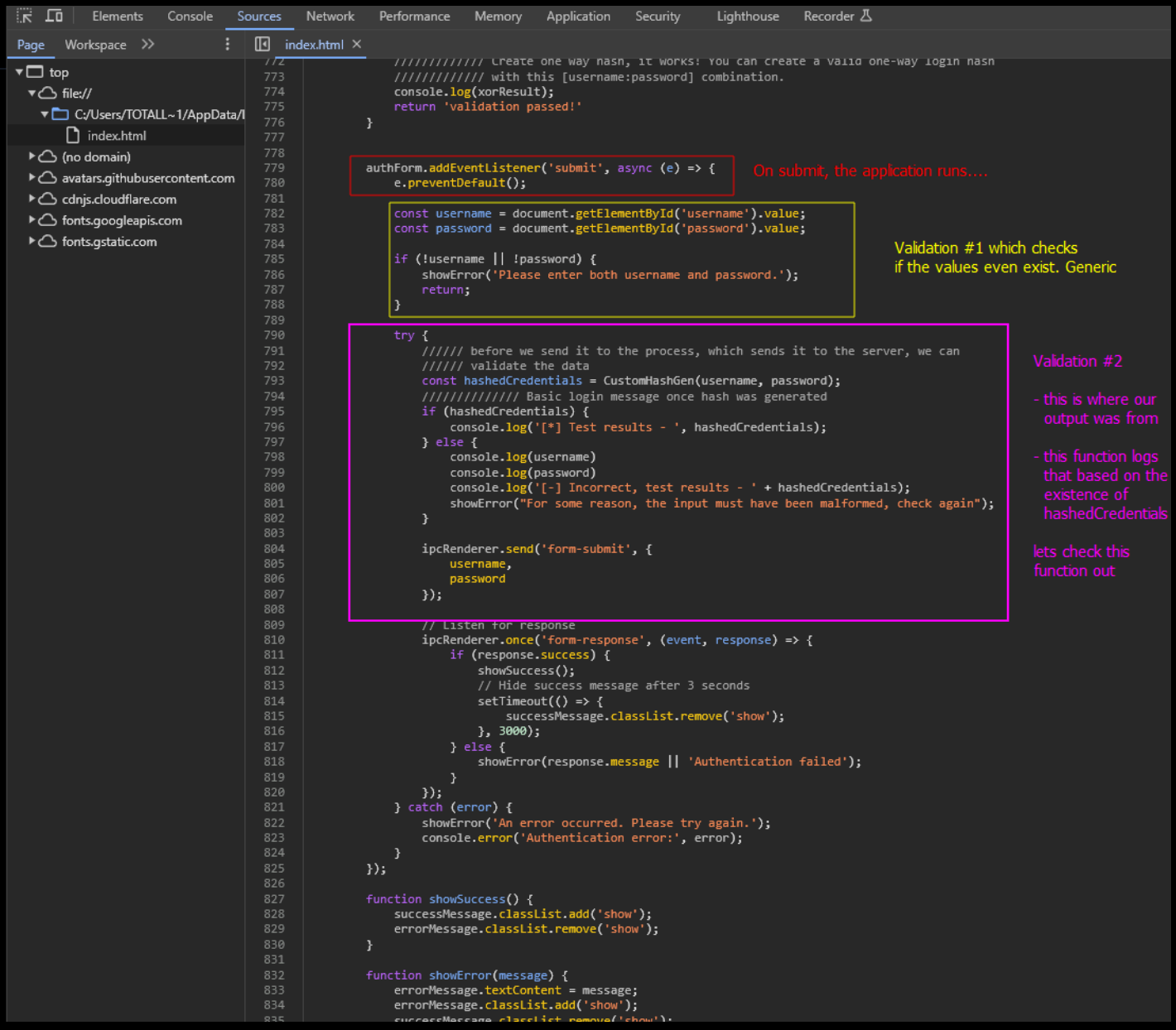
.....well that did not work out too well, the process just kills itself. Lol. Lets try again. This time.

Hitting Enter with 1 input of all 'A's, which ends up yielding...



Oh yay, some real output in the dev console we can follow. Let's check it out so we can analyze the demo.

- Note: Click on the link, index.html, referenced in the dev console, in relation to the output '[*] Test Results - validation passed'. This will bring you to the snippet below.



Checking out the function `CustomHashGen` we see the following.

```
745   @routine : CustomHashGen
746                Runs a username and password string through a custom
747                hash generator to generate a one-way hash.
748   */
749         function CustomHashGen(username, password) {
750             // Create a combined string
751             const combined = username + ':' + password;
752
753             ////// rule states that username must all be lowercase
754             ////// if they are not, we return. Basic standard
755
756
757             /////////// XOR op for each charactter uses this key
758             const xorKey = 'SkyPenguinSecretKey';
759             let xorResult = '';
760
761             /////// XOR Op
762             for (let i = 0; i < combined.length; i++) {
763                 /////// This standard is weird, it makes sure the user can not have 0s
764                 /////// in your password only. It exits, silently, no warning, just,
765                 /////// ....silence. More of a reason for somebody to look at it lol
766                 if (combined.charCodeAt(i) == 48) {
767                     ipcRenderer.send('e');
768                 }
769                 const charCode = combined.charCodeAt(i) ^ xorKey.charCodeAt(i % xorKey.length);
770                 xorResult += String.fromCharCode(charCode);
771             }
772             ////////////// Create one way hash, it works! You can create a valid one-way login hash
773             ////////////// with this [username:password] combination.
774             console.log(xorResult);
775             return 'validation passed!'
776         }
777
```

Concats user and password with delim ':'

Missing code? possibly backend referenc

KEY!

XOR LOOP

<- main proprietary routine

Validation version only, does not fully run the hash routine but reveals enough

Dang! What a routine to walk through! Seriously. This code has some beef. But given the scenario, it's expected when one tries to copy the structure of a hash routine from their backend onto their frontend for simple validation sake.

As you can see, this code clearly exposes the hardcoded logic that theoretically would be used on the server as well to store passwords and usernames, making the standard useless on the backend because the frontend just slaps it in there for every client to see!

While the function does not reveal the full resulting hash, its not that hard for anybody to guess, all they would need to have is additional information in relation to the application or system they are looking at.

The mistake is simple, the devs should not have used frontend validation here. It was not a matter of user experience at all, in fact, UX should not have even been thought of when considering the use of a proprietary algorithm.

`> How could the developer, though, improve something SIMILAR for UX?`

Well, without indirectly revealing the internal structure, the client can use various techniques, specific to the data the standard requires, such as validating the fact that the password and username may only contain a specific set of characters or numbers, instead of just quitting the application (*as we saw when we use an input with 0.*)

- Did you get stuck? `I` did not reference this. But if you look at the function, and just read it, the for loop states that, as i is equal to 0, if `I` is less than the length of the combined string, then increment `I` by one. This is literally just iterating over characters. When you look at the second condition, `` `if combined.CharCodeAt` `` you can see its referencing the code 48 in decimal, on an ASCII chart, which is `'0'` (*since we are dealing with string characters in a web application*). If its 0, it sends 'e' to the application, presumably, 'e' means 'exit' which calls the main rendering process within the Electron app to close the application. Because every time the input contains a 0, anywhere, anyway, it exists.

Since this is such a basic fix, there are only a few select other options and most of them don't fit within this context. So let's move onto our second application demo.

Scenario 2 - app validates phone number stored in localStorage before use

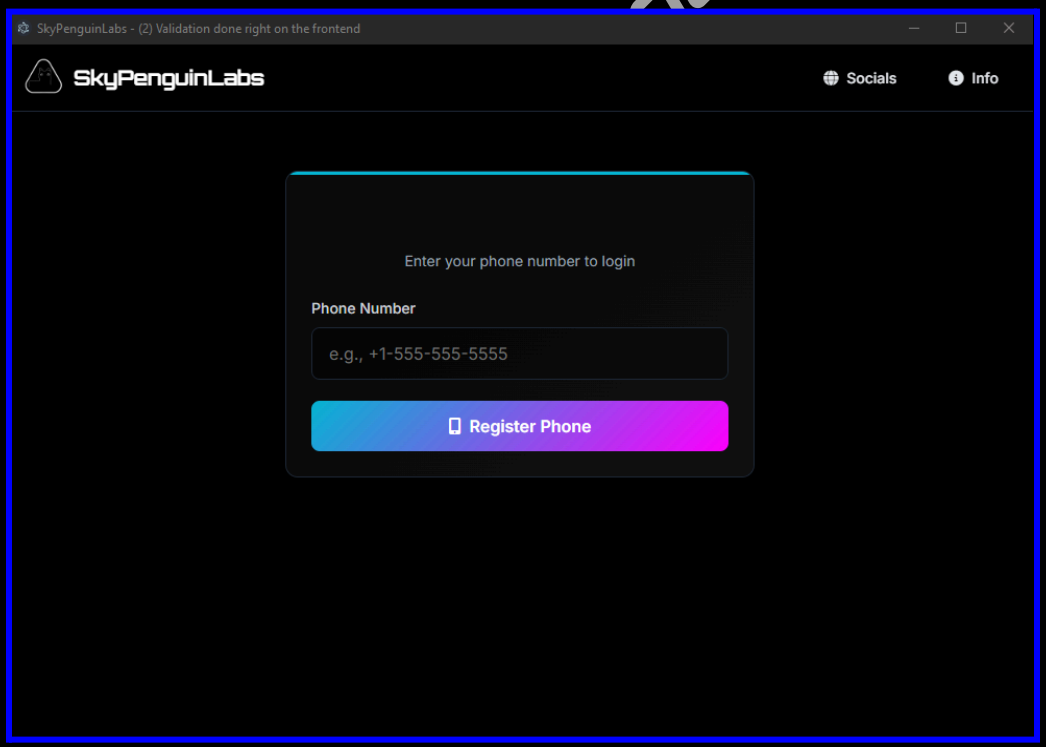
In this scenario, an application takes a phone number from the user, validates it, and then stores it in `localstorage` for use later. When the time comes, the program will pull the phone number from localstorage and validate it again before sending it to the location where it needs to be sent.

The reason this is considered a ***good*** practice (*__NOT__ to be mistaken with a 'secure' practice*) is because if something in the application affects the phone number stored in `localstorage` before the app sends it to the server, the client might return an unexpected error if the data was not of the correct format.

Typically this would also be a server side thing, and if the data was sent anyway, the client is designed to handle the error from the server anyway saying server-side validation failed.

However, it's still good to make sure the data is valid before sending it to the server on the client anyway if you can as it prevents the useless requests to the server.

When you download the application from here and then run it, you end up getting shown the following.



As you can see, it's simply just a phone number input dialog.

While I did not provide a devtools console for you to peak into the code yourself, I will show you a snippet of the backend for you to pay attention to which helps you identify this.

Internally, when the form calls the process to take in the user input phone number, the program calls the following

```
form submission
phoneForm.addEventListener('submit', (e) => {
    e.preventDefault();

    const phone = document.getElementById('phone').value;
    localStorage.setItem('userPhone', phone) //// store unvalidated
    ///// get the phone

    ////// some process in between that does not access local storage
    ipcRenderer.send('w');

    ipcRenderer.once('finished', () => {
        ///// validate and register the number
        ipcRenderer.send('validate-phone', {
            phone
        });
    })
```

Storing here is okay as long as the process below does not use the number prior to validation

Once ipcRenderer.send -> w() is finished, it validates and registers the phone number

As you can see, this code is mainly fine. The only danger area is that components inside of whatever function 'w' is, will get impacted if the number is not of the correct format.

Though, we ignore this, and we continue on when the main Electron process sends back a response. The continuation specifically ends up validating the number, and registering it if it is properly formatted.

If not, the form catches the response from the main electron process, and handles accordingly.

```
    ipcRenderer.once('validate-response', (event, response) => {
        if (response.success) {
            showRegSuccess(response.message); //// validation was good, so was registration, main process
            /// ended up using the phone number and sending it to a server once registration was valid
        } else {
            showRegError(response.message);
            localStorage.removeItem('userPhone'); //// remove when validation fails (good)
        }
    });
});
```

As its pretty easy to tell, the point of this brick of control flow is to check the response from the validation/registration function, which returns whether or not the number can be used in this application entirely.

This simply just removes it from local storage after showing an error if the number is `__NOT__` of the correct format. If it is, it continues control flow after showing the registration was a success.

And just like that, we have a simple method of frontend validation that is implemented purely for the sake of saving both the servers and the developers (*even users*) time without breaking.

Scenario 3 - Engaging AI for Validation & Vibe Coding

We will talk about this in an upcoming article/lesson, but vibe coding is always bad. No matter who does it, the whole terminology (*Vibe coding is an AI-assisted software development practice where a user provides natural language prompts to an AI model, which then generates the functional code, reducing the need for the human to write code line-by-line*) is just bad.

While we do admit that AI can be used correctly, vibe coding is a direct and *incorrect* use of AI/ML models.

The reason?

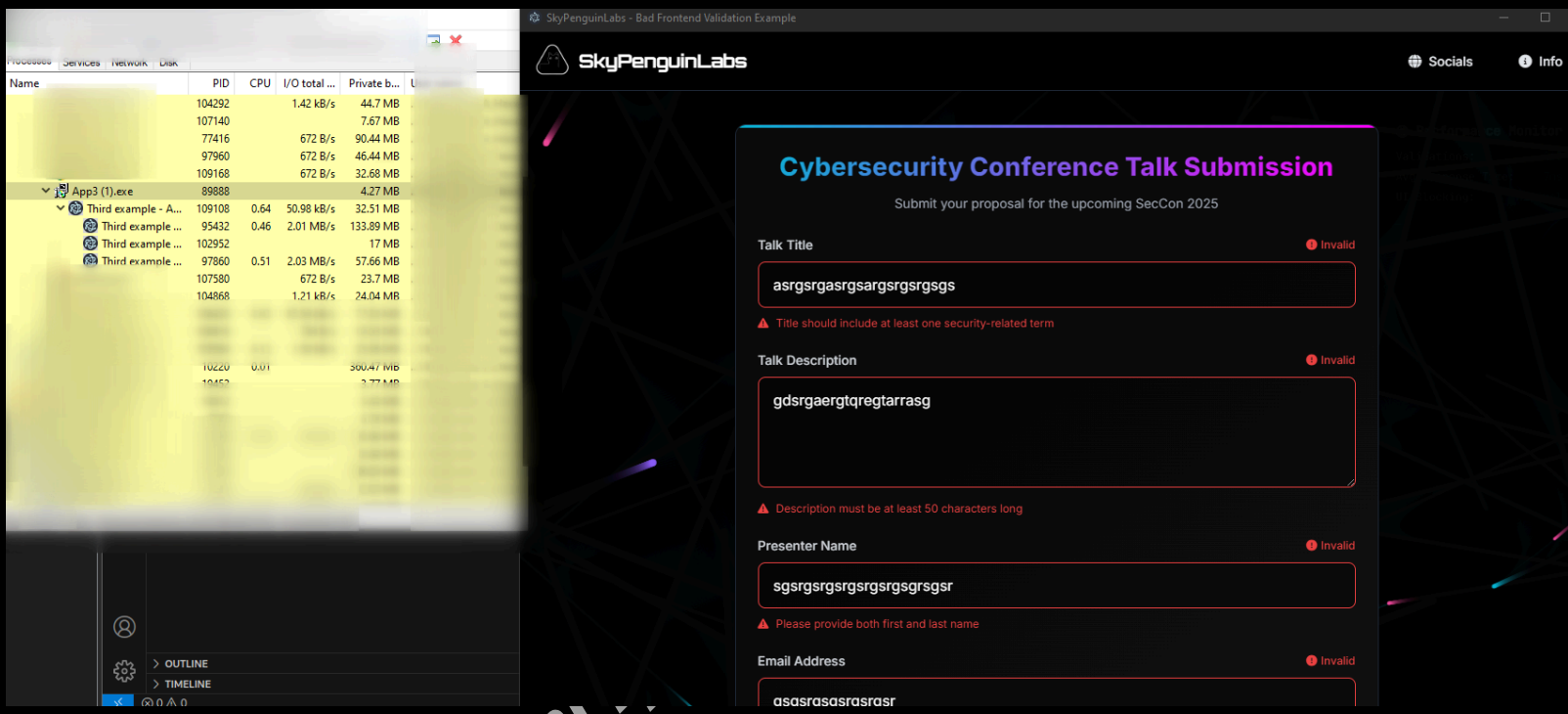
Let's consider an example.

We have some developers who were told a frontend application needs frontend validation NOW! The purpose behind this was to ensure the apps performance was going to be better, and that there would not be so many users flooding up their ticketing system, wondering why they got random JS errors that overflow because they used the wrong symbol in an input field, and the client does not handle such edge cases.

These developers, naturally, are put under a painstaking timeline. 2 days.

So they naturally go to AI to save them. The result is- well, correct. By direct instructions, but is not really- correct by implementation.

Lets check it out, here.



The idea is to run a process monitor such as procmon or process hacker alongside of this application, and as you begin to type, make sure you check out how the performance of the application does. Do you notice anything drastically different?

You should notice, that unlike most applications, when throwing a few characters at this app, it's not handled the best, and if you are on a laptop, is even bound to lag a bunch!

Why? Well, if we look at the products code on the next page (which contains a screenshot of said code), we may be able to see the reason why it eventually becomes problematic.

```
function validateTalkTitle(title) {
    let isValid = true;
    let reasons = [];

    for (let i = 0; i < 10000; i++) {
        const reversedTitle = title.split('').reverse().join('');
        // reverse to scan
        if (reversedTitle === title && title.length > 3) {
            // set const to true
            const palindromeCheck = true;
        }
    }

    // Check if length < 10
    if (title.length < 10) {
        isValid = false;
        reasons.push('Title must be at least 10 characters long');
    }

    // Check if length > 100
    if (title.length > 100) {
        isValid = false;
        reasons.push('Title must be less than 100 characters long');
    }

    // Check for common security conference buzzwords
    const buzzwords = ['hack', 'cyber', 'security', 'exploit', 'vulnerability', 'threat', 'breach', 'attack'];
    let hasBuzzword = false;
```

That being said, just looking at it is enough to actually tell what some of the problem areas are.

And yes, while this is one brick of code out of the 300 that exist within that codebase, this is just one good sample that really makes a twisty turn on the internal performance of the app.

AI could also have been used to make this better, but instead, because of a pressured set of devs, and a set of people that just want features, it's enough to end this slopped up mess of WTF.

Conclusion

This article was originally going to be paid, and intended to be much longer and in-depth, but it almost made no sense. This is a topic that is covered enough and is not much in our niche. So we figured we would just put it out there for fun, and use it as a chance to build ElectronJS applications we may pick apart for fun.

In this article, we explored the basics of understanding of frontend validation and how little people try to pass it off as something that can work as a security mechanism, but how many people also forget about it as a useful tool for user experience (*UX*).

While we are not primarily designers, we did have to build a website, and in this lesson, we did explain some methods as to how UX can aid in the overall *consumer/user* experience.

We finally walked through some various examples of both good and bad practices that can be representations of the obvious.

To which we finally end here, with me thanking you for taking advantage of our free courses, and asking for just one more moment of your time.

SPL, or SkyPenguinLabs, runs on community members like you- who drive our attention and views through taking advantage of what we give back to the community: i.e-free courses. If you would like, ALL digital content is on etsy, we have our main site here, and if you are not looking to spend money, it would be appreciative if you mention this course to somebody you know! Even one friend, or GPT is enough! :D

We hope to see you again- and genuinely, thanks for reading a course that felt so slopped together we made it free LOL! A lesson learned in the book case on generalized topics.