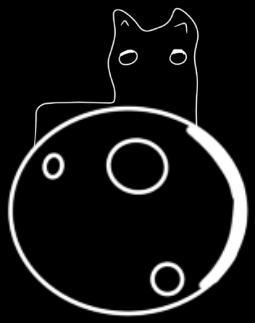


SkyPenguinLabs

Samples

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 0x04** | 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.

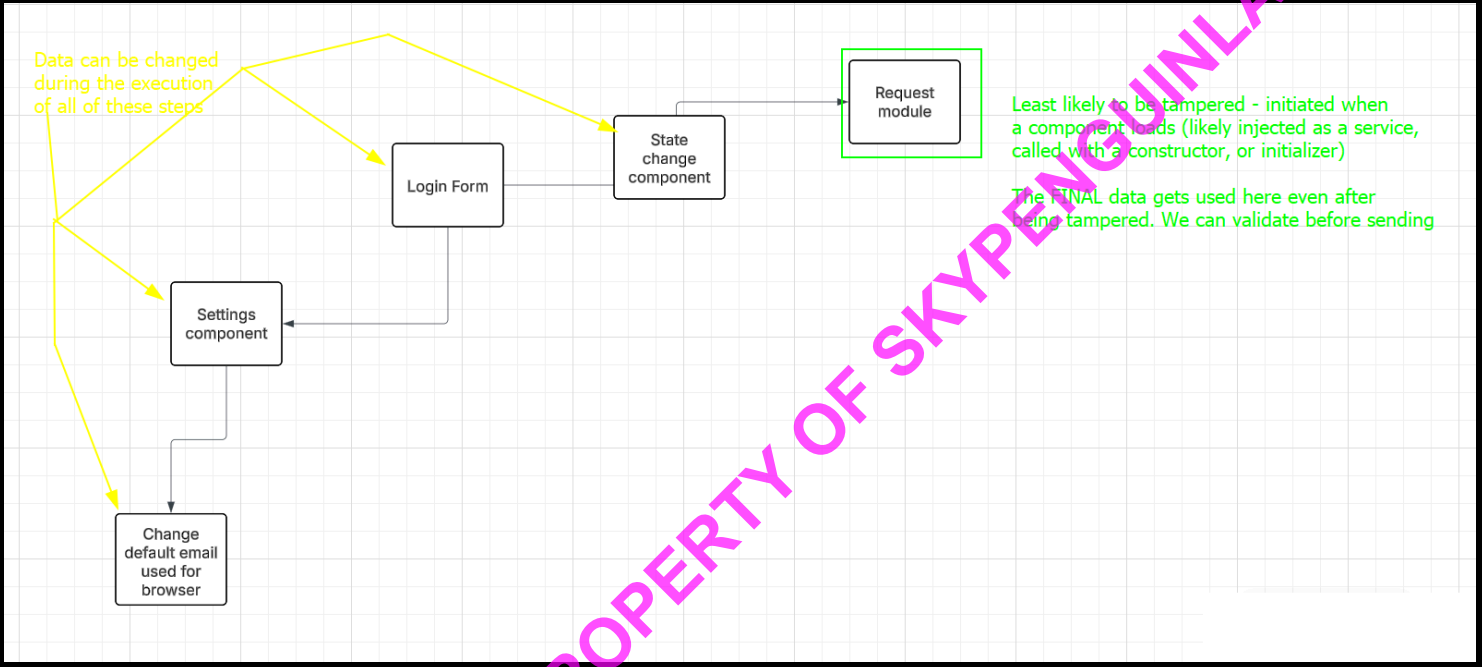
This component

- **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.