IMPLEMENTANDO VALIDAÇÃO DE INFORMAÇÕES EM UM FORMULÁRIO COM VUE.JS

mpsxdeveloper
https://github.com/mpsxdeveloper

Resumo

Nesse artigo será demonstrado a criação de algumas regras de validação de dados informados por um usuário em um formulário na web. O formulário faz parte de uma aplicação desenvolvida com Vue.js e as validações serão feitas sem o uso de biblioteca específica para esse fim.
Palavras-chave: Dados. Validações. Formulário. Vue.js

Abstract

This article will demonstrate the creation of some validation rules for data entered by a user in a web form. The form is part of an application developed with Vue.js and validations will be done without using a specific library for this purpose.
Keywords: Data. Validations. Form. Vue.js

Introdução

Validar as informações fornecidas pelo usuário em sistemas na web é de fundamental importância para garantir o bom funcionamento das regras do sistema. Não é recomendado pressupor que o usuário vai conseguir preencher todos os campos do formulário com informações precisas e condizentes com o que foi solicitado, é necessário que o sistema forneça o máximo de detalhes para o preenchimento e verifique informações incorretas retornando para o usuário onde devem ser feitas as correções, garantindo que somente informações corretas sejam enviados ao servidor.

Algumas considerações sobre validação de dados

A validação de dados informados pelos usuários se tornou parte essencial no desenvolvimento de aplicações web, de modo que para facilitar esse processo, diversas bibliotecas e ferramentas foram desenvolvidas somente para esse propósito. Embora sejam úteis e bastante recomendado utilizar essas ferramentas automatizadas de validação, é importante que o desenvolvedor tenha uma noção básica dessa etapa e saiba filtrar as informações fornecidas pelo usuário final sem o uso dessas ferramentas, ainda que em modelos simples de sistemas.

O formulário utilizado e estrutura do projeto

Para os propósitos de testes, criamos um diretório com o nome validation-vuejs (nome do projeto) e uma página index.html com um formulário HTML simples com os campos: nome, e-mail, CPF, senha, confirmação de senha e um botão de salvar. Para lidar com o preenchimento, a validação dos dados e feedback para o usuário, foi utilizado Vue.js (Options API) via CDN. Uma folha de estilos

CSS foi incluída para melhorar um pouco a parte visual da aplicação. O script app.js será responsável pela lógica do Vue.js e a estrutura final do projeto ficará como na Figura 1:

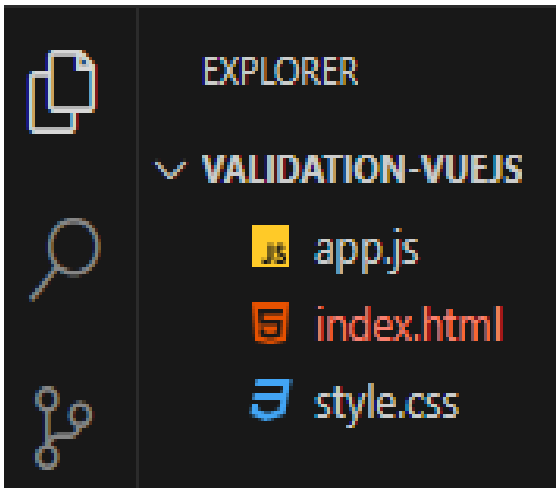

Figura 1: Arquivos do projeto

A página index.html deve ter a seguinte codificação:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <script src="https://unpkg.com/vue@3/dist/vue.global.js"></script>
    <link rel="stylesheet" href="style.css">
    <title>Validation Vue.js</title>
</head>

<body>
    <div id="app">
        <form @submit.prevent="validate">
            <h1>Registro</h1>
            <input type="text" v-model.trim="name" placeholder="Nome" maxlength="30">
            <input type="text" v-model.trim="email" placeholder="E-mail" maxlength="50">
            <input type="text" v-model.trim="cpf" placeholder="CPF" maxlength="11">
            <input type="password" v-model="password" placeholder="Senha" maxlength="60">
            <input type="password" v-model="confirm" placeholder="Confirmar Senha"
maxlength="60">
            <button type="submit">Salvar</button>
        </form>
        <div v-show="errors.length" class="messages" ref="messages">
            <ul>
                <li v-for="(error, i) in errors" :key="i">{{ error }}</li>
            </ul>
        </div>
```

```
    </div>
    <script src="app.js"></script>
</body>

</html>
```

No cabeçalho da página carregamos o Vue.js via CDN e também a folha de estilos CSS. No final da página, carregamos o código JavaScript app.js que será responsável pela lógica de nossa aplicação. Para o formulário, temos inputs do tipo texto e password, e um botão do tipo submit para enviar os dados para o backend. Note que o Vue.js já está sendo utilizado no formulário HTML, pois utiliza-se o método *@submit* para envio do formulário, com o modificador *prevent* para permitir que tratemos os dados antes que os dados sejam enviados, ou seja, modificamos o comportamento padrão de enviar os dados automaticamente para o servidor, pois queremos verificar e testar as informações digitadas pelo usuário.
Em *v-model*, associamos cada campo do formulário com uma variável no Vue.js, de modo que podemos sincronizar os valores informados nos inputs e suas atualizações.
Depois do formulário temos uma div com um *v-show="errors.length"* que é uma renderização condicional do Vue.js, pois essa div só será exibida caso existam erros na variável do tipo array *errors*. Essa div também tem um atributo especial *ref="messages"*, que faz com que se tenha uma referência ao elemento HTML correspondente, nesse caso, o div que exibe as mensagens. Fizemos isso porque queremos mudar a estrutura desse div eventualmente com o Vue.js, sem ter que utilizar JavaScript com querySelector, por exemplo. Uma lista com os erros ocorridos serão exibidos dentro de uma tag li do HTML através do loop *v-for*, fazendo com que todas as mensagens de erros fiquem centralizados em único lugar. Outra possibilidade seria exibir individualmente cada erro abaixo do input que gerou o erro, mas para facilitar o código, optou-se pela estratégia anterior.
Em *v-model.trim*, o modificador trim vai retirar os espaços em branco do começo e fim das strings dentro dos inputs automaticamente, portanto mesmo que existam apenas espaços nos inputs nome, cpf e e-mail, ainda assim serão considerados campos vazios. Nos inputs do HTML temos o atributo *maxlength* estipulando o máximo de caracteres permitidos, ou seja, antes do Vue.js ser invocado, já estamos fazendo alguma verificação da integridade dos dados no próprio formulário.
Para o arquivo *style.css* o código será o seguinte:

```
#app {
    width: 500px;
    max-width: 800px;
    margin: 0 auto;
    background-color: #03047A;
    padding: 20px;
    border-radius: 8px;
}

form {
    display: flex;
    flex-direction: column;
}
```

```
input[type="text"], input[type="password"], button {
    margin: 5px 0;
    padding: 10px;
    border-radius: 8px;
    border: none;
    outline: none;
}

button {
    border: none;
    background-color: #90ee90;
    cursor: pointer;
}

.messages {
    margin-top: 15px;
    padding: 2px;
    background-color: #DC143C;
    color: #fff;
    font-size: 18px;
}

.success {
    background-color: #90ee90;
    color: #00f;
}

h1 {
    color: #fff;
    text-align: center;
}
```

Todos esses estilos aplicados ao app, formulário e mensagens são bem simples, o que vale ressaltar aqui é o estilo aplicado à classe .success, pois essa classe não está presente em nenhum elemento na página HTML. Essa classe será adicionada dinamicamente ao div de mensagens pelo Vue.js caso o formulário seja submetido com sucesso, ou seja, não foi detectado nenhum problema em nossas validações.
Para o script *app.js* temos o seguinte código:

```
const { createApp } = Vue

const app = createApp({
    data() {
        return {
            name: '',
            email: '',
```

```
        cpf: '',
        password: '',
        confirm: '',
        emailPattern: /^[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]
+)*$/,
        errors: [],
        errorMessages: ['Nome é obrigatório', 'Nome deve ter entre 5 e 30 caracteres',
          'E-mail é obrigatório', 'Formato de e-mail incorreto', 'CPF é obrigatório',
          'CPF deve ter 11 números, sem espaços, letras ou símbolos', 'Senha deve ter no mínimo 8
caracteres',
          'Confirmação de senha deve ter no mínimo 8 caracteres', 'As senhas informadas não são
iguais'
        ]
      }
    },
    methods: {
      validate() {
        this.errors = []
        if(this.name === '') {
          if(!this.errors.includes(this.errorMessages[0])) {
            this.errors.push(this.errorMessages[0])
          }
        }
        if(this.name !== '' && this.name.length < 5 || this.name.length > 30) {
          if(!this.errors.includes(this.errorMessages[1])) {
            this.errors.push(this.errorMessages[1])
          }
        }
        if(this.email === '') {
          if(!this.errors.includes(this.errorMessages[2])) {
            this.errors.push(this.errorMessages[2])
          }
        }
        if(this.email !== '' && !this.emailPattern.test(this.email)) {
          if(!this.errors.includes(this.errorMessages[3])) {
            this.errors.push(this.errorMessages[3])
          }
        }
        if(this.cpf === '' || this.cpf.length !== 11) {
          if(!this.errors.includes(this.errorMessages[4])) {
            this.errors.push(this.errorMessages[4])
          }
        }
        if(this.cpf.length === 11) {
          let hasErrors = false
          for(let i = 0; i < this.cpf.length; i++) {
```

```
                const numbers = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
                if(!numbers.includes(this.cpf.charAt(i))) {
                    hasErrors = true
                    break
                }
            }
            if(hasErrors && !this.errors.includes(this.errorMessages[5])) {
                this.errors.push(this.errorMessages[5])
            }
        }
        if(this.password === '' || this.password.length < 8) {
            if(!this.errors.includes(this.errorMessages[6])) {
                this.errors.push(this.errorMessages[6])
            }
        }
        if(this.confirm === '' || this.confirm.length < 8) {
            if(!this.errors.includes(this.errorMessages[7])) {
                this.errors.push(this.errorMessages[7])
            }
        }
        if(this.password.length > 7 && this.confirm.length > 7 && this.password !== this.confirm)
{
            if(!this.errors.includes(this.errorMessages[8])) {
                this.errors.push(this.errorMessages[8])
            }
        }
        if(this.errors.length === 0) {
            this.$refs.messages.classList.add("success")
            this.errors.push("Formulário enviado com sucesso")
            this.name = this.email = this.cpf = this.password = this.confirm = ''
        }
    }
  }
})

app.mount('#app')
```

No começo utilizamos a função *createApp* para criarmos nosso aplicativo, que recebe como parâmetro os dados e métodos e outros itens que podem ser utilizados pelo Vue.js, e no final, montamos nosso app com *mount('#app')*, onde *#app* corresponde ao id da div principal na nossa página HTML.
Antes de verificarmos minuciosamente esse código Vue.js, se clicarmos no botão Salvar, as validações serão feitas, e teremos o feedback sobre o que há de errado quanto ao preenchimento dos campos, demonstrado na Figura 2:

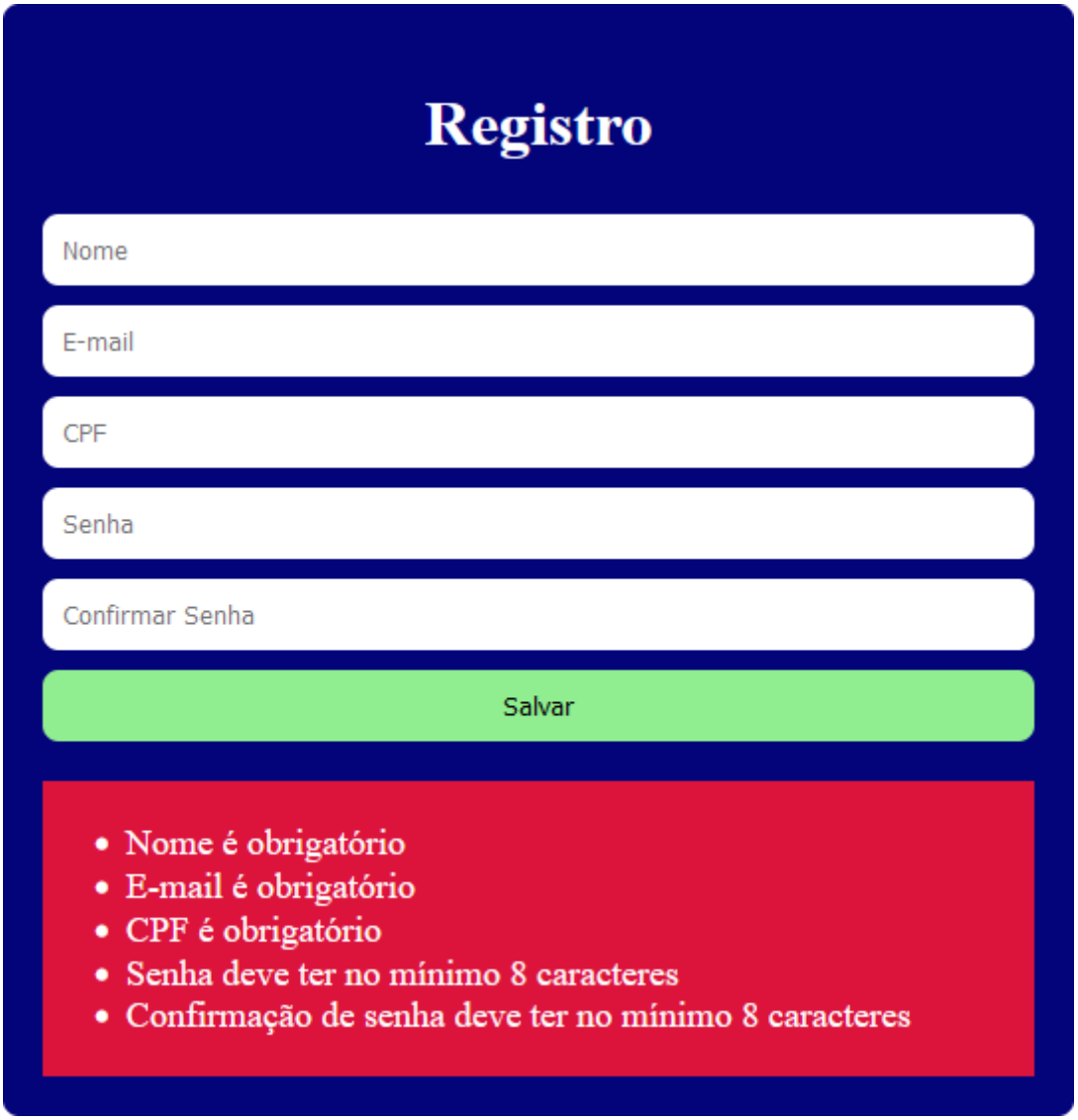

Figura 2: Feedback da validação do formulário

Entendendo melhor o código, em data() configuramos as variáveis que queremos tratar em nosso app, sendo assim, name, email, cpf, password e confirm correspondem aos atributos v-model do nosso formulário. Cada atualização nos inputs do formulário farão com que esses valores sejam atualizados em sincronia.
A propriedade *emailPattern* contém a expressão regular que é utilizada para validar se o usuário informou um e-mail válido. Não basta apenas que o usuário informe um e-mail, é necessário checar se é um formato válido de e-mail.
Em *errors: []*, temos o array responsável por guardar as mensagens de erros de cada input preenchido de forma incorreta, de modo que é feito um loop na página HTML exibindo os itens desse array. A diretiva *v-if="errors.length" verifica* se tem items no array de erros, então a página vai exibi-los.
Com o array errorMessages, guardamos todas as mensagens de erros possíveis no nosso aplicativo, bastando indicar o índice desse array para exibir a mensagem correspondente ao erro.
Em *methods*, configuramos nosso método *validate()*, que é onde ocorrem todas as validações dos campos. Antes de tudo, a variável *errors* é configurada para um array vazio novamente com *this.errors = []*, fazendo com que a cada clique no botão salvar, as mensagens antigas de erros não

sejam acumuladas, ou seja, só vão reaparecer se o erro que gerou a mensagem se repetir. Esse método *validate()* vai ser chamado na página após o clique do botão Salvar, pois por se tratar de um botão do tipo submit, vai invocar *@submit.prevent="validate"* do formulário.
A linha *if(this.name === '')* checa se o nome foi preenchido, caso contrário, a mensagem de erro correspondente deve ser adicionada ao array *errors* com *this.errors.push(this.errorMessages[0])*, porém, antes devemos fazer uma checagem para verificar se essa mensagem já se encontra inserida nesse array com o seguinte teste: *if(!this.errors.includes(this.errorMessages[0]))*. Em seguida, se o nome foi preenchido, é testado se a quantidade de caracteres é menor do que 5 ou maior que 30, caso isso ocorra, teremos uma mensagem correspondente para esse tipo de erro.
Essa é a lógica para todos os testes, variando apenas no que deve ser testado antes de prosseguir com o fluxo do código. Para fins didáticos, as mensagens retornadas pelo sistema às vezes agrupam várias possibilidades de erros cometidos pelo usuário em relação a um campo específico, porém numa aplicação real em ambiente de produção, o ideal seria que cada erro tivesse uma mensagem bem específica para que o usuário pudesse corrigir mais precisamente onde errou.
Prosseguindo, com o campo e-mail, além de checarmos se foi preenchido, também checamos se é um e-mail com um formato válido no teste *!this.emailPattern.test(this.email)*, onde utilizamos o valor salvo na expressão regular *emailPattern* como parâmetro de teste.
Com o campo CPF, dois testes iniciais são feitos, se foi preenchido e se tem exatamente 11 caracteres. Se o input passar por esses dois testes, o próximo passo é testar se todos os 11 caracteres informados do CPF são somente números.
A última etapa é testar os campos de senha e confirmar senha. Primeiro é testado se cada um deles foi preenchido e foi informado a quantidade mínima de caracteres exigida, que aqui nesse caso são 8. Então, num último teste, se os campos senha e confirmar senha foram preenchidos com 8 ou mais caracteres, é comparado se os valores são iguais, se não forem iguais, o usuário será alertado desse erro.
Se preenchermos o formulário com todas as informações corretas (dentro do que é exigido no nosso código Vue.js, o formulário terá seus campos reiniciados e aparacerá uma mensagem de que o formulário foi enviado com sucesso). Com *this.$refs.messages* (recurso template refs do Vue.js) temos acesso ao elemento HTML correspondente ao div do formulário que exibe as mensagens, portanto podemos adicionar dinamicamente a classe *.success* a esse div. Com *this.errors.push* incluímos uma mensagem confirmando o envio do formulário, fazendo com que o Vue.js exiba a div de mensagens. Embora não tenha erros, isso é feito para que o Vue.js exiba a div de mensagens (pois essa div só é exibida se o array *errors* contem itens). Na Figura 3, vemos a mensagem correspondente no caso do formulário ter passado por todas as validações com sucesso:

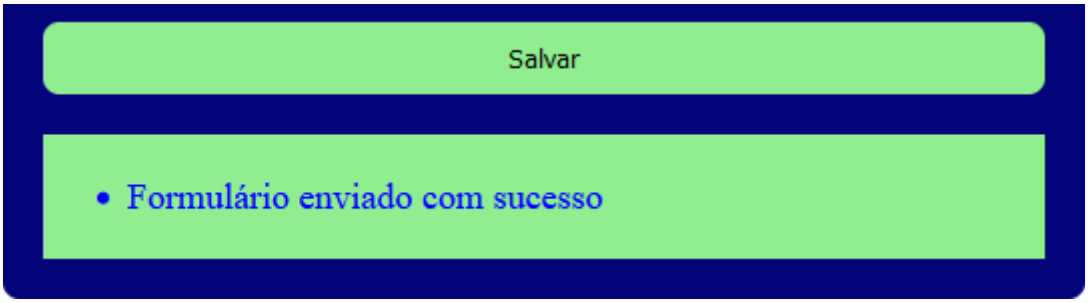

Figura 3: Formulário com as validações testadas com sucesso

Ao preenchermos o formulário com as seguintes informações: nome com menos de 5 caracteres, e-mail com formato incorreto, CPF com 10 dígitos e uma letra (ou espaço ou símbolos), senha e confirmar senha com 8 caracteres ou mais, porém diferentes uma da outra, nota-se que as mensagens de erro serão diferentes das mensagens de erros anteriores, com textos mais específicos quanto aos tipos de erros, como é demonstrado na Figura 4:

Figura 4: Feedback diferente dependendo dos erros

Considerações finais

A validação das informações fornecidas pelos usuários é um processo que já faz parte do desenvolvimento e manutenção de aplicativos e sistemas, garantindo a melhor confiança e resultados para se trabalhar com os dados e se obter os resultados previstos nas regras de negócio da aplicação. O desenvolvedor deve ter consciência da importância da validação dos dados, tanto no lado cliente quanto no lado servidor, e buscar a melhor solução para cada tipo de sistema. Em sistemas com poucos campos, como o que foi demonstrado nesse artigo, uma validação utilizando o próprio Vue.js com JavaScript simples foi suficiente para validar todas nossas regras de negócio, entretanto, conforme for aumentando a complexidade do sistema e quantidade de informações a

serem preenchidas, deve-se considerar a adoção de bibliotecas específicas para essa finalidade, tais como Vuelidate e VeeValidate, que são bibliotecas de validação do ecossistema Vue.js.

Bibliografia

Vue.js. Disponível em: <https://vuejs.org/>. Acesso em: 15/07/2024.