DETALHES IMPORTANTES AO FAZER FORMULÁRIOS:

SEMPRE VALIDE OS CAMPOS

- Campos obrigatórios devem ter o “required”;

USE LABELS PARA FACILITAR A VIDA DOS USUÁRIOS

- Sempre amarre o “for” do label aos ids que correspondem ao input, isso faz com que quando o cliente clicar no label ele acione o campo o input;

USE OS TIPOS CERTOS

- Não esqueça de ver se os campos realmente estão atendendo ao tipo certo por através do atributo “type”;
- Por exemplo não permita que o um campo de email seja “type = text” ele têm que ser “type = email”;

USE O TITLE PARA DAR DICAS

- O atributo title dá uma dica aos usuários sobre oque eles devem escrever dentro de um determinado campo;
- Ele funciona de 2 formas, em campos sem “required” se o usuário colocar o mouse encima do campo aparece um tooltip com a dica sobre o campo. Em campos com “required” ele vai dar a dica quando o usuário não preencher o campo;

LIDANDO COM CAMPOS INVÁLIDOS DE MANEIRA EFICIENTE

- Todo input é um objeto Javascript que possuí uma porção de par chave e valor. Uma destas chaves é a chave “validity” que possuí uma chave dentro dela chamada “valid”, quando um campo não é validado por através do “required”, a chave “valid” recebe o valor de “false”, e quando ele é validado, ela recebe o valor de “true”. Isso abre para nós um leque de possibilidades para formatação de inputs quando um campo não é validado.
  Por exemplo, podemos gerar uma classe que é acionada sempre que o “valid” for “false” para adicionar comportamentos ao nosso input, usando classList.remove(‘nome da classe’) ou adicionar adicionar uma classe quando o “required” é satisfeito com “classList.add(‘nome da classe’);

CUSTOMIZANDO MENSAGENS DE ERRO

- É possível também customizar as mensagens de erro que aparecem no formulário (aquelas mensagens que aparecem no tooltip quando o required não é satisfeito), isso é possível por através de chaves que existem dentro do objeto input, por exemplo:
  - **valueMissing:** Recebe como valor uma mensagem que será exibida no tooltip quando o required não é satisfeito;
  - **typeMismatch:** Recebe como valor uma mensagem que será exibida no tooltip quando um valor for colocado de forma errada, por exemplo um email num input do type “email” que não tenha o “@”;
  - **patterMismatch:** Recebe como valor uma mensagem que será exibida no tooltip quando um valor não combinar com um determinado padrão pré-definido, como um padrão ReGex por exemplo;

- **customError:** Recebe como valor uma mensagem que será exibida no tooltip quando um erro que nós mesmos customizamos acontecer. Para customizar erros usamos o método “setCustomValidity” no Javascript para os nossos objetos input;

SOBRE DATAS E FAIXA ETÁRIA

- Em alguns formulários é necessário cumprir uma determinada faixa etária para que a pessoa possa preencher o formulário;
- Use essa fórmula para faixa etária:

```
const dataNascimento = document.querySelector(#id_do_input)

dataNascimento.addEventListener(‘blur’, (e) => {
    validaDataNascimento(e.target)
}

function validaDataNascimento(input){
    const dataRecebida = new Date(input.value)
    let mensagem = ‘’

    if(!maiorQue18(dataRecebida)){
        mensagem = ‘Você deve ser maior que 18 anos para se
cadastrar!’
    }

    input.setCustomValidity(mensagem) //Exibe uma mensagem de erro
                                      //customizada à nossa escolha
}                                     //se um valor vazio for passado
                                      //o js entende que a validação
                                      //deu certo

function maiorQue18(data) {
    const dataAtual = new Date()

    const dataMais18 = new Date(data.getUTCFullYear() + 18,
data.getUTCMonth(), data.getUTCDate())

    return dataMais18 <= dataAtual
}
```

Note na fórmula acima que nós usamos um evento blur sempre que o usuário termina de digitar a sua data de nascimento a função validaDataNascimento pega o valor do input da data chama a função maiorQue18 para comparar se a data de nascimento + 18 anos á frente é menor ou igual a data de hoje, se a quantidade de anos for superior a data de hoje significa que o usuário ainda não completou 18 anos, e uma mensagem será mostrada para ele.

SOBRE SENHAS

- Delimite um número mínimo para os caracteres de senha por através do “minlength = “6”” por exemplo;
- Delimite um número máximo para os caracteres de senha por através do “maxlength = “12”” por exemplo;
- Utilize o atributo “pattern” para definir um método de escrita que deve ser empregado, geralmente usamos uma regex para isso, dessa forma:

  “pattern = “^(?=.*[a-z])(?=.*[A-Z])(?=.*[0-9])(?!.*[^!%*¨/\()=§]).{6,12}””

Essa é uma regex determina que a senha tenha pelo menos 1 letra maiúscula, 1 numeral e pode conter qualquer caractere especial que não seja esses: espaço^!%*¨/\()=§;

SOBRE VALIDAÇÃO DE CPF

É obvio que não há forma melhor de validar CPFs do que usando a validação de uma API, porém também podemos fazer a validação na mão, podemos fazer isso da seguinte forma:

- O CPF deve seguir a seguinte regra:
  Todo CPF têm 2 dígitos verificadores ao final, após o traço do CPF, para cada um deles é feita uma conta diferente para chegar ao dígito verificador:
  - **1º Dígito verificador:**
    Equivale a subtração de 11 pelo resto da divisão de 11 pela soma do 1º dígito do CPF multiplicado por 10, mais o 2º dígito do CPF multiplicado por 9... e por ai vaí até o 9º dígito do CPF multiplicado por 2:

    Digamos que o CPF fosse 123.456.789-??
    Teríamos que fazer:

```
soma = (1*10)+(2*9)+(3*8)+(4*7)+(5*6)+(6*5)+(7*4)+(8*3)+(9*2)
```

    Depois temos que dividir esse valor por 11 e pegar somente o resto da divisão e fazer o seguinte teste condicional:
    - Se o resto da divisão for menor que 2, o dígito verificador deverá ser 0 (zero);

```
restoDaDivisao = soma % 11
If (restoDaDivisao < 2)
    return 0
```

    - Se o resto da divisão for igual ou maior que 2, deveremos subtrair o resto da divisão por 11, e vamos chegar ao nosso dígito verificador;

```
restoDaDivisao = soma % 11
If (restoDaDivisao >= 2)
    return 11 - restoDaDivisão
```

  - **2º Dígito verificador:**
    Equivale a subtração de 11 pelo resto da divisão de 11 pela soma do 1º dígito do CPF multiplicado por 11, mais o 2º dígito do CPF multiplicado por 10... e por ai vaí até o 10º dígito do CPF – isso mesmo, o segundo dígito vai incluir o 1º dígito verificador na soma - multiplicado por 2:

    Digamos que o CPF fosse 123.456.789-9?
    Teríamos que fazer:

```
soma = (1*11)+(2*10)+(3*9)+(4*8)+(5*7)+(6*6)+(7*5)+(8*4)+(9*3)+(9*2)
```

    Depois temos que dividir esse valor por 11 e pegar somente o resto da divisão e fazer o seguinte teste condicional:
    - Se o resto da divisão for menor que 2, o dígito verificador deverá ser 0 (zero);

```
restoDaDivisao = soma % 11
If (restoDaDivisao < 2)
    return 0
```

- Se o resto da divisão for igual ou maior que 2, deveremos subtrair o resto da divisão por 11, e vamos chegar ao nosso dígito verificador;

```
restoDaDivisao = soma % 11
If (restoDaDivisao >= 2)
    return 11 – restoDaDivisão
```

- **Para facilitar podemos usar as seguintes funções para fazer essa validação de CPF:**

```
document.querySelector('#cpf').addEventListener('blur', e =>
validaCPF(e.target))

//Função chamada...
function validaCPF(input){
    //troca tudo o que não for dígito por nada...
    const CPFformatado = input.value.replace(/\D/g, '')
    let mensagem = ''

    if(!checaCPFRepetido(CPFformatado) ||
!checaEstruturaCPF(CPFformatado)){
        mensagem = 'O CPF digitado não é valido!'
        input.setCustomValidity(mensagem)
    } else {
        alert('CPF VÁLIDO!')
        input.setCustomValidity(mensagem)
    }


}

function checaCPFRepetido(cpf){
    const valoresRepetidos = [
        '00000000000',
        '11111111111',
        '22222222222',
        '33333333333',
        '44444444444',
        '55555555555',
        '66666666666',
        '77777777777',
        '88888888888',
        '99999999999',
    ]

    let CPFValido = true

    valoresRepetidos.forEach(valor => {
        if(valor == cpf)
            CPFValido = false
    })

    return CPFValido
}

function checaEstruturaCPF(cpf){
    const multiplicador = 10
```

```
    return checaDigitoVerificador(cpf, multiplicador)
}

function checaDigitoVerificador(cpf, multiplicador){

    //faz a função não ser chamada recursivamente infinitamente
    if(multiplicador >= 12){
        return true
    }

    let multiplicadorInicial = multiplicador
    let soma = 0
    const cpfSemDigitos = cpf.substr(0, multiplicador -
1).split('')
    for(let contador = 0; multiplicadorInicial > 1;
multiplicadorInicial--){
        soma = soma + cpfSemDigitos[contador] *
multiplicadorInicial
        contador ++
    }

    const digitoVerificador = cpf.charAt(multiplicador - 1)
    if(digitoVerificador == confirmaDigito(soma))
        return checaDigitoVerificador(cpf, multiplicador + 1)

    return false

}

function confirmaDigito(soma){
    let restoDaDivisao = soma % 11
    if(restoDaDivisao >= 2){
        return 11 - restoDaDivisao
    } else {
        return 0
    }
}
```

PARA VALIDAÇÃO DE CEP

Para valição de CEP temos uma baita ajuda que é a API da via CEP, podemos acessá-la pela URL: https://viacep.com.br/ws/cep-a-ser-passado-sempre-somente-letras/json
Podemos acessá-la por através de uma fetch passando a URL e um campo de opções para podermos capturar os valores daquele determinado cep.

Uma função que podemos usar é:

```
function recuperaCEP(input){
    const cep = input.value.replace(/\D/g, '')
    const url = `https://viacep.com.br/ws/${cep}/json`
    const options = {
        method: 'GET',
```

```
        mode: 'cors',
        headers: {
            'content-type': 'application/json;charset=utf-8'
        }
    }

    if(!input.validity.patternMismatch && !input.validity.valueMissing){
        fetch(url, options)
            .then(response => response.json())
            .then(data => {
                if(data.erro){
                    input.setCustomValidity('Não foi possível buscar
CEP')
                    return
                }
                input.setCustomValidity('')
                preencheCamposComCEP(data)
            })
    }

}

function preencheCamposComCEP(data){
    const logradouro = document.querySelector('[data-tipo="logradouro"]')
    const cidade = document.querySelector('[data-tipo="cidade"]')
    const estado = document.querySelector('[data-tipo="estado"]')

    logradouro.value = data.logradouro
    cidade.value = data.localidade
    estado.value = data.uf
}
```

Para ver o conteúdo completo, acesse a pasta de html5>tags consultas rapidas>formularios>formularios para teste

USE MÁSCARAS MONETÁRIAS

Quando lidamos com inputs que envolvem dinheiro, podemos usar uma máscara – máscaras de input são programações que aplicam um determinado formato aos valores que são colocados dentro de input, no exemplo das máscaras monetárias seria colocar o Cifrão do dinheiro, ponto e vírgula nas casas decimais e assim por diante – onde o usuário iria colocar somente números.

Uma grande ajuda para aplicar máscaras monetárias é uma biblioteca chamada simple-mask-money, podemos importá-la usando o comando:

```
npm i simple-mask-money –save
```

Existe um link externo também, mas no presente momento ele está indisponível, que seria o link:

```
<script src="https://github.com/codermarcos/simple-mask-
money/releases/download/v3.0.0/simple-mask-money.js"></script>
```

Para ver como aplicamos isso na prática acesse o formulário criado na pasta html5 na parte de forrmulários para máscaras monetárias, o código de implementação foi esse:

No HTML:

```
<body>
    <form action="">
        <label for="money">
            Preço
            <input inputmode="numeric" value="0,00" id="money">
        </label>
        <input type="submit">
    </form>
</body>
<script src="validaMONEY.js" type="module"></script>
<script src="./../../../node_modules/simple-mask-money/lib/simple-mask-money.js"></script>
```

No arquivo validaMONEY.JS:

```
let input = document.getElementById('money')

SimpleMaskMoney.setMask(input, {
    prefix: 'R$ ',
    fixed: true,
    fractionDigits: 2,
    decimalSeparator: ',',
    thousandsSeparator: '.',
    cursor: 'end'
})
```

No objeto que a função setMask recebe os valores que vemos no objeto são para os seguintes objetivos:

prefix: para criar um prefixo antes do valor monetário
fixed: não sei ainda, mas o valor sempre é true
fractionDigitis: para fracionar o número de dígitos após a casa decimal dos centavos
decimalSeparator: para escolher um sinal de separação para as casas decimais dos centavos
thousandsSeparator: para escolher um sinal para a separação de casas centenais

DATA-ATTRIBUTES EM VEZ DE IDS

- Em vez de ficar criando uma série de ids para cada input, crie um único data-attribute e crie variáveis que poderão ser referenciadas no javascript. Dessa forma:

```
data-um_nome_qualquer_a_sua_escolha =
“nome_da_variavel_que_referencia_o_input”
```

Explicando o código: data-attributes são atributos que podemos criar para gerar uma variável que captura os valores dos inputs e poderá ser referenciada por através de um nome à nossa escolha, elas devem obedecer a regra de ter um prefixo “data-” seguido por um nome à nossa escolha, e devem receber como valor um nome que identifique aquele input.

data-identificador = “nome_que_identifica_o_input”

Os data-attributes são muito úteis para identificarmos vários inputs pelo atributo invés de id, e como cada um deles recebe um nome que o identifica, podemos criar um array em JS para adicionar um comportamento específico para cada um, como por exemplo:

```
//Uma constante que pega todos inputs e os coloca num array...

const inputs = document.querySelectorAll('input')

//Adiciona um evento blur para validação de cada input capturado...

inputs.forEach(input => {
    input.addEventListener('blur', (event) => {
        validaInput(event.target) //cada evento chama a função
    })                            // validaInput
})

//A função validaInput verifica se o input passado possuí um valor de
data-attribute, se tiver, ele pegará uma função específica para aquele
valor data-attribute...

function validaInput(input){
    const tipoDeInput = input.dataset.tipo_de_data-attribute //dataset

    //chamamos um objeto que pode conter uma determinada chave
    //contendo uma função de validação como seu valor

    if(validadores[valor_do_data_attribute]){
        validadores[valor_do_data_attribute](input)
        //aqui estamos chamando a função para aquele determinado
        //data-attribute e passando o evento como parâmetro
    }
}

//Objeto que contém uma função de validação diferente para cada valor
de data-attribute que existir...

const validadores = {
    valor_do_data_attribute1 = input => funcaoValidadora(input),
    valor_do_data_attribute2 = input => funcaoValidadora(input),
    valor_do_data_attribute3 = input => funcaoValidadora(input)
}
```

SERVIDORES PARA TESTES:

- Caso deseje usar servidores para testar o seu formulário você pode usar:
  - Apache
  - Live-Server do VSCode
  - Browser-Sync
- Lembrando que o Browser-Sync é uma biblioteca Node que pode ser instalada direto no Node usando o comando:

  ```
  npm i –g browser-sync
  ```

  Caso a instalação não dê certo, utilize o sudo, assim:

  ```
  sudo npm i –g browser-sync
  ```

Depois de aberto, digite o comando:

```
browser-sync start –s –f . –directory
```

Esse comando faz com que o servidor seja startado, será apresentado no terminal qual a porta que o localhost vai utilizar para o servidor no browser.