Aula Node

# JavaScript Assíncrono (AJAX)

## O que é ?

Chegamos num tópico muito importante, um estudo um pouco mais profundo do JavaScript Assíncrono, esse que possibilita os sites modernos serem muito mais “user friendly” devido à dinâmica oferecida por esse comportamento.

AJAX é um acrônimo para *Asynchronous JavaScript and XML*, **JavaScript Assíncrono e XML**, essa é uma técnica de desenvolvimento Web que permite a criação de aplicações mais interativas. Apesar do nome, não existe a necessidade de ser em XML, pois a hoje em dia o JSON é o mais utilizado. Um dos principais objetivos é tornar as respostas das páginas Web mais rápidas pela troca de pequenas quantidades de informações com o servidor Web, nos bastidores.

Além disso, evita-se que a página Web inteira tenha que ser recarregada cada vez que alguma nova informação precisa ser consultada no servidor. Em geral, isso significa que páginas Web com **recursos AJAX** permitem maior interatividade, velocidade de processamento e usabilidade.

## Como desenvolver utilizando AJAX

Bem, primeiramente, as funções assíncronas podem ser usadas tanto no front-end quanto no back-end, mas elas são mais comuns no front-end pelos motivos vistos anteriormente. Uma função assíncrona é declarada utilizando a *keyword* **async**, isso é necessário pois o JavaScript, por padrão, funciona **sincronamente**, então precisamos dizer à ele quando uma função não é síncrona, ou seja, quando ela é assíncrona.

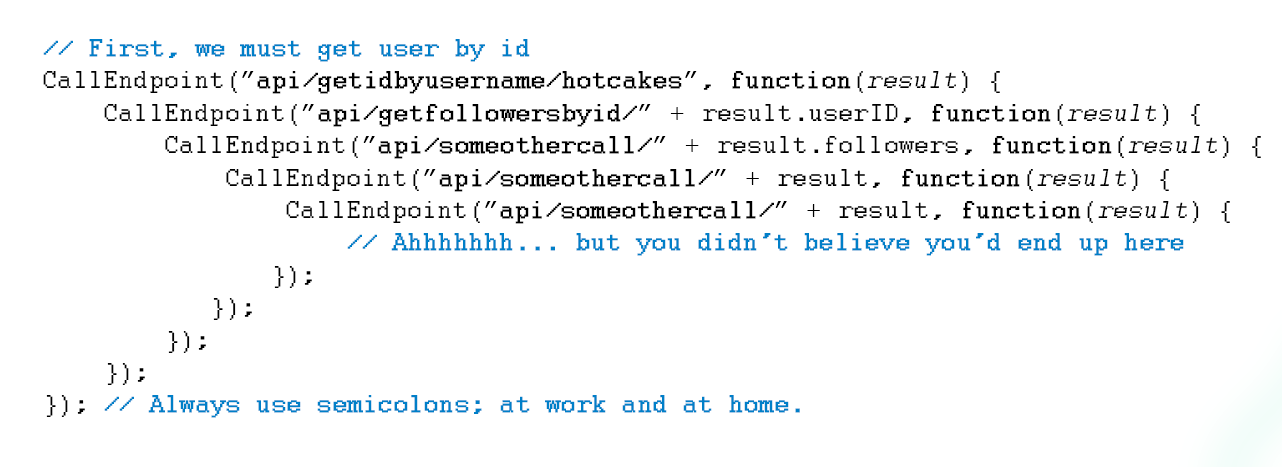
Quando uma função assíncrona é chamada, ela retorna uma [Promise](https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise). Quando a função assíncrona retorna um valor, a Promise será resolvida com o valor retornado. Quando a função assíncrona lança uma exceção ou algum valor, a Promise será rejeitada com o valor lançado. (Calma, já já veremos o que são essas Promises)

Dentro de uma função que foi declarada utilizando a *keyword* async, podemos utilizar outra expressão, que é o **await**. Esta expressão nos permite pausar a execução da função enquanto espera outra função terminar. Em termos mais técnicos, esta expressão espera pela resolução da Promise passada, e depois retoma a execução da função assíncrona e retorna o valor resolvido.

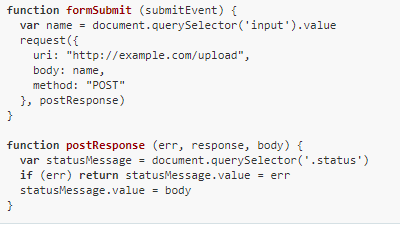
Hoje em dia utiliza-se o método global ***fetch*** para realizar requisições assíncronas. O ***fetch*** pode nos retornar um sucesso ou um erro, respectivamente é o que chamamos de ***Promisse Resolved*** e  ***Promise Reject***, e caso dê um erro / reject, o código não irá quebrar. Além disso, é possível pegar / “capturar” esse erro utilizando um bloco de ***try / catch***.

Veja esse exemplo:

## Um pouco de história…

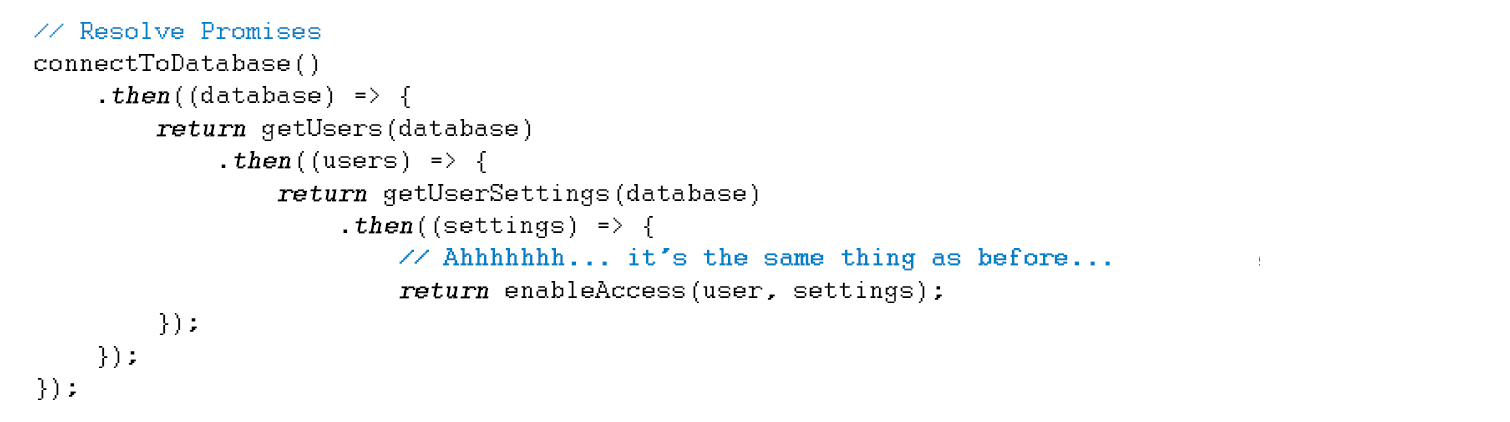
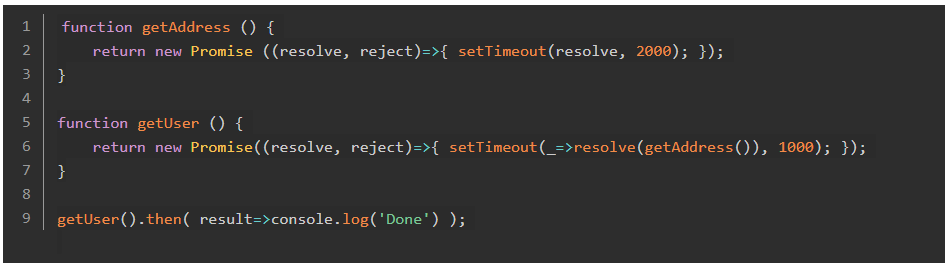
 Até agora, o desenvolvimento de nossos programas tem sido de forma síncrona com o JavaScript, utilizando o que chamamos de **callback**, e não tem problema algum nisso, porém, para certas coisas o uso de callbacks pode ser algo muito ruim para se administrar, por exemplo:

A imagem acima é um exemplo do que chamamos de ***Callback Hell***. O objetivo dessa aplicação seria acessar a API do GitHub e quando o servidor deles nos respondesse, realizar alguma lógica, trabalhar com os dados, e coisa do tipo, e logo em seguida, realizar outra requisição… e assim por diante.

 Por bastante tempo as coisas eram feitas assim, claro, existem técnicas para deixar o código acima mais limpo, mais fácil de ler e de realizar manutenção, através de técnicas como: nomear as funções anônimas, Modularizar as funções, tratar adequadamente dos erros e etc… Não existe muita necessidade de demonstrar isso, mas fiquem com um exemplo:

Para solucionar isso, criaram o que chamamos de Promises (promessas), uma Promise pode ser resolvida ou rejeitada, além de podermos encaixar funções para serem executadas assim que forem terminarem através do encadeamento das palavras chaves *then* e *catch*, resolvendo o problema do Callback Hell. Exemplo de Promises:

Entretanto, isso nos trouxe outro problema, a “***Promise Hell***”:

Bem, nosso estudo no curso está mais focado ao uso do async/await, pois ele é uma “evolução” da promise. Como pode ser visto, mesmo a Promise que deveria ser a “solução” do Callback Hell, ela ainda está “incompleta”... O uso do async/await é uma ótima solução, pois: Nós, como seres humanos, temos dificuldade de pensar assincronamente, as coisas fazem mais sentido quando estão em sequência. Por isso a abordagem do async/await é muito boa, ele mantém a leitura do código de uma maneira síncrono e mantém assincronicidade da lógica.

**Nota:** O uso de async/await funciona apenas em métodos/funções que retornam uma promise. Por exemplo: O método **fetch** realiza uma requisição ao servidor e nos retorna uma Promise com todas as informações que o servidor respondeu. Essa promise sempre será **resolved**, a não ser se ocorrer um erro com a comunicação com o servidor. Isso significa que mesmo o servidor retornando um status 404 (ou outro tipo de status), a promise ainda será resolvida (ou seja, não entrará no bloco catch).

Material para auxiliar:

* <https://medium.com/trainingcenter/entendendo-promises-de-uma-vez-por-todas-32442ec725c2>
* <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Global_Objects/Promise>
* <https://braziljs.org/artigos/promises-no-javascript/>
* <https://medium.com/@js_tut/the-great-escape-from-callback-hell-3006fa2c82e>

Vídeos para auxiliar:

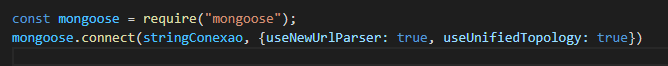
* <https://www.youtube.com/watch?v=gB-OmN1egV8&list=PLqrUy7kON1meuCvGp2D6yTglZhPTT_s_f&index=4&t=1262s> (inglês)
* <https://www.youtube.com/watch?v=mDbA3EZme4c> (português)

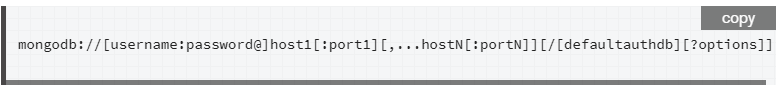
# Conectando no MongoDB Atlas

Finalmente iremos aprender a salvar nossos dados no banco de dados. No curso veremos, como dito anteriormente, um banco de dados não relacional, o MongoDB. Podemos fazer isso de duas maneiras: *remotamente* ou *localmente*. Em ambos os casos iremos utilizar o pacote ***mongoose***. Ele irá nos auxiliar para fazer essa comunicação.

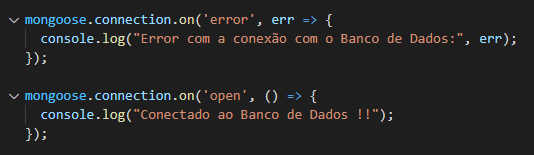
Primeiramente, instale-o

Em seguida, você deve chamar o pacote no seu *app.js* ou equivalente e então, se conectar ao servidor através de sua *string de conexão*.

O objeto, passado como segundo parâmetro, é essencial para evitar mensagens de aviso do mongoose no console

Mas o que é essa “*string de conexão*” ? Bem, é uma string que possui os dados necessários para nosso aplicativo se comunicar com nosso banco de dados, nele possui informações sigilosas como: login, senha, endereço do host, porta, coleção... Este é seu formato genérico, não se assuste, não é tão difícil como parece:

Não podemos agora nos esquecer da parte mais importante quando estamos trabalhando com Desenvolvimento Web, que é o tratamentos de erros ! Vamos observar se a conexão ao Banco vai ocorrer com sucesso, ou se terá um erro.

O interessante dessas linhas de código, é que não observamos apenas o erro de quando ocorre a conexão. Mas também caso a conexão caia durante o uso da aplicação. Isso ocorre pois o mesmo evento “*error*” é disparado em ambas as situações.

## Localmente

Para nos conectarmos ao Banco de Dados localmente, precisamos que o *MongoDB Community Server* esteja instalado

Os passos anteriores são válidos tanto para a etapa local quanto para a etapa remota. Para nos conectar ao banco local, devemos passar a seguinte string de conexão:

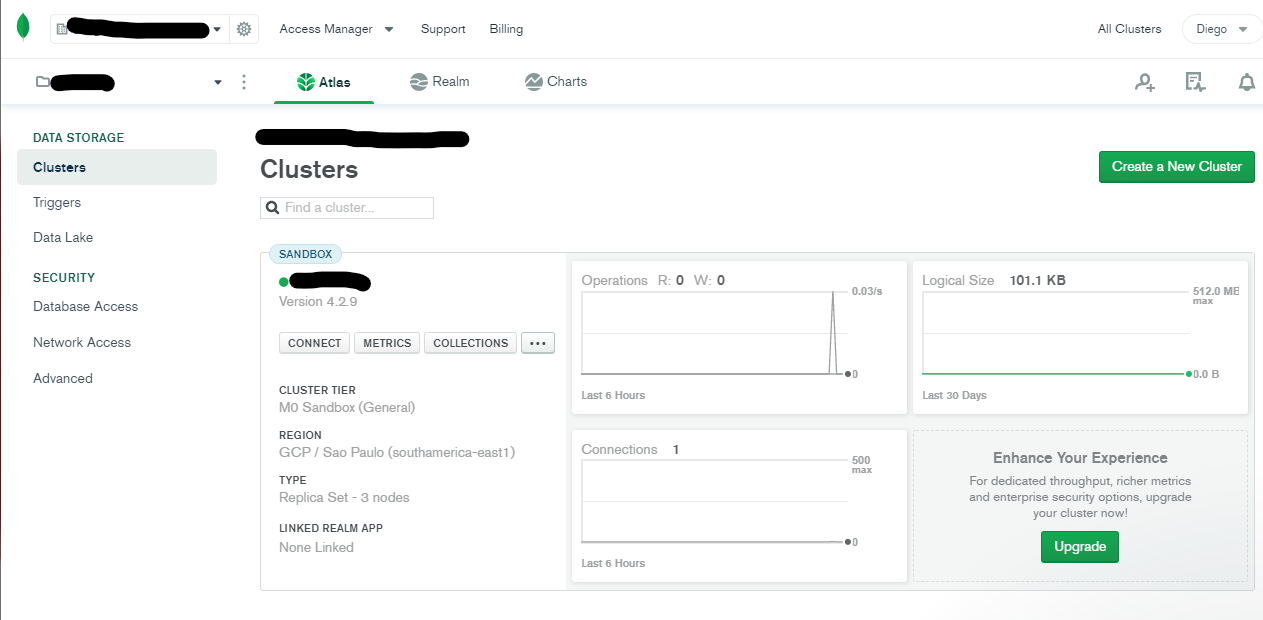
 Essa string é formada pelo: “*mongodb”*, que é fixo, pelo ip do servidor, no caso, localhost, depois pela porta, e por último, o nome da coleção que estaremos trabalhando (Caso ela não exista, o *mongoose* irá cria-la automaticamente).

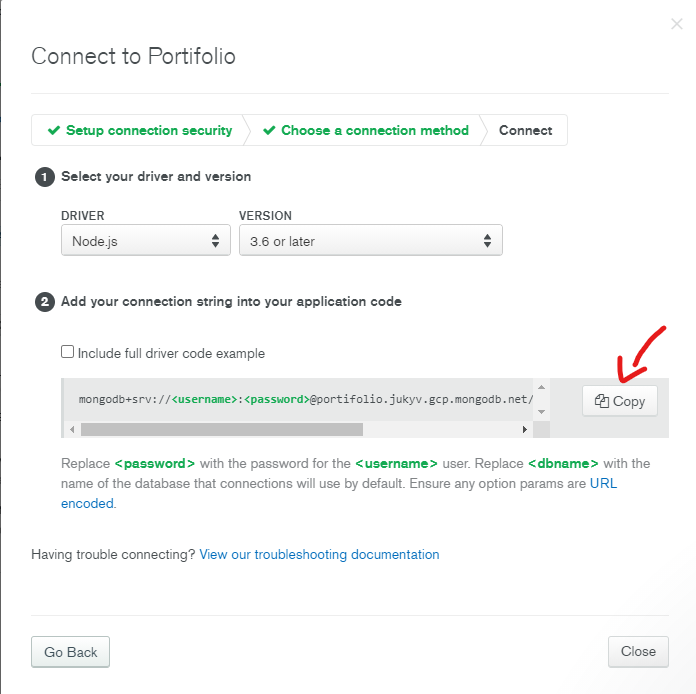
Como estaremos nos conectando em um banco pertencente ao nosso próprio computador, não há necessidade de inserir dados de “*login*”.

## Remotamente

Caso queira acessar um Banco de Dados remoto, que esteja hospedado em um servidor, devemos primeiro escolher um serviço para isso. No curso, iremos trabalhar com ferramentas gratuitas, no caso, iremos utilizar os próprios serviços que o MongoDB oferece.

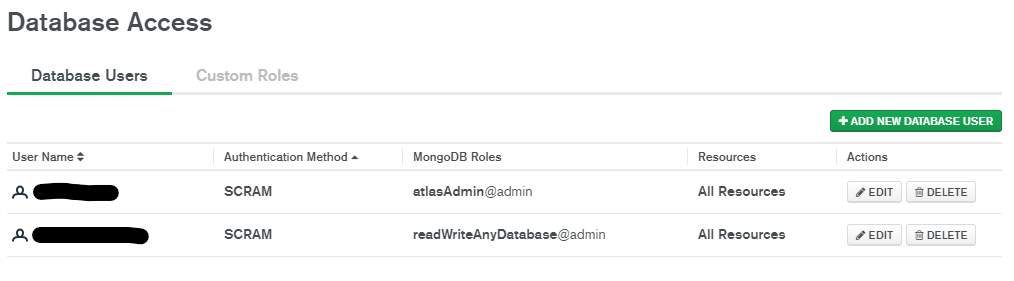
Para começar, é necessário criar uma conta no [site](https://www.mongodb.com/) deles. Em seguida, criar um novo *Cluster* e selecionar o plano gratuito. Feito isso, uma tela parecida com essa deve aparecer:

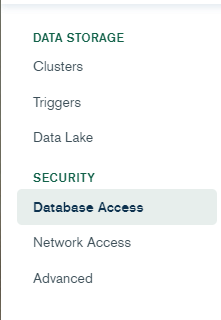
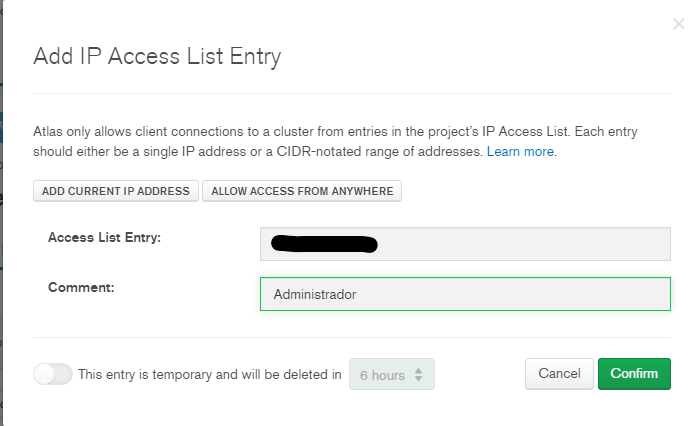
 Clique em *connect* > *connect your application* e copie a *connection string* que ele te apresentar.

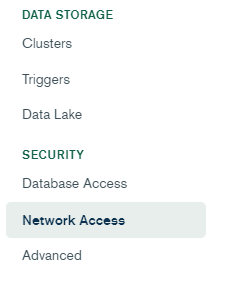
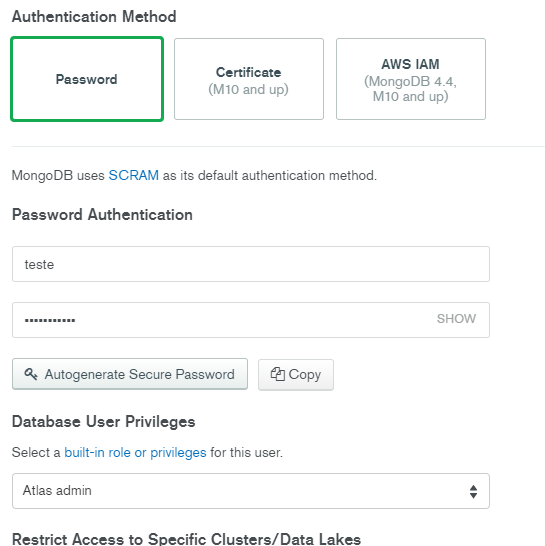
Haverá três campos para serem preenchidos: *username, password* e *dbname*. Você pode pensar que o *username* e *password* são seus dados de acesso à sua conta MongoDB, mas **não** são ! Eles na verdade, são as credenciais de acesso de um usuário para o seu Cluster.

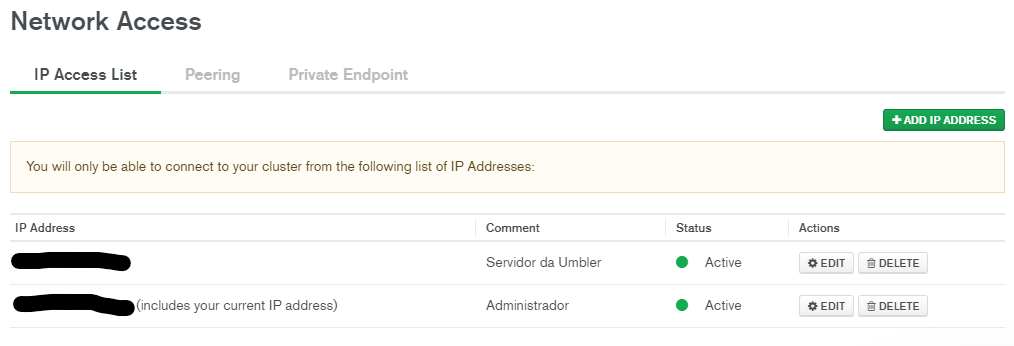
Para entendermos melhor, imagine que há mais de um servidor conectado ao seu banco de dados. Talvez, nem todos possam ter as mesmas permissões que o administrador, você pode querer que um servidor apenas leia as informações e outro que leia e grave....

Bem, para fazer isso, devemos clicar no menu lateral *Database Access*, logo abaixo de *Security*. Agora, devemos criar um novo usuário, clique no botão verde *Add New Database User*, e preencha os campos necessários.



Em seguida, devemos dizer ao MongoDB qual/quais máquinas tem permissão de acessar nosso Cluster. Clique agora em *Network Access*, abaixo de *Network Access*, adicione um novo Ip, clicando no botão verde, clique em *Add Current Ip Address*, coloque um comentário, como Administrador, e confirme.



Os próximos passos serão vistos em aula, mas são válidos para ambos os casos.

# Validando Dados

A validação dos dados é uma das coisas mais importante na web. Ela serve para proteger sua aplicação e seu servidor de códigos maliciosos. Voce deve inserir validação sempre que houver um campo de preenchimento de dados em sua página, como um formulário, ela pode ser *client-side* ou *server-side*.

O ideal é ter ambas, mas veremos no curso apenas validações client-side (há muitos tutoriais e pacotes disponíveis na internet para te ajudarem a fazer validação *server-side*).

O ideal é ter ambas as validações, pois se você tiver apenas no lado do cliente, estará sujeito a falhas de segurança, pois o cliente poderia apenas desabilitar o JavaScript e *voilá*, inserir o que quiser no formulário.

Enfim, agora que entendemos isso, vamos começar. Podemos fazer isso de duas maneiras, escrever nosso próprio código JavaScript para validar ou utilizar o poder do HTML5 ao nosso favor. Como no inicio do curso já fizemos a primeira opção, ensinarei agora a segunda.

Para fazer isso é bem simples, podemos apenas inserir os atributos *required*, *maxlength*, *minlenth*, *type* e entre outros. Voce pode ver todas os atributos disponíveis [aqui](https://developer.mozilla.org/pt-BR/docs/Web/HTML/Element/input). Por exemplo, um campo onde a pessoa deve preencher com um email deve ter o *“type=email”* no atributo do elemento, e se ele não pode estar vazio, o atributo *required* irá se certificar disso.

O único lado ruim disso é que as mensagens são muito padronizadas, é muito difícil fazer algo mais customizado. Mas assim como as validações server-side, também há muitos tutoriais mostrando como fazer algo mais único.