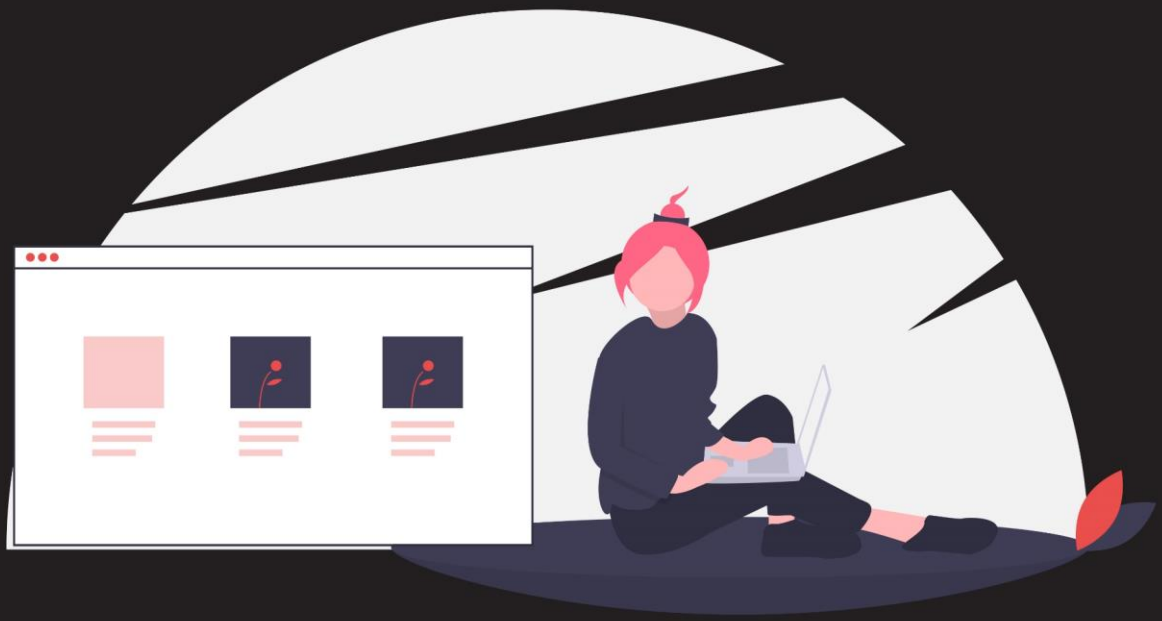
# JavaScript

## A linguagem da web

Andrey Araujo Masiero



# Sumário

# Introdução

JavaScript é uma linguagem nativa da Web. Ela é utilizada para dar dinamismo e interatividade nas páginas feitas com linguagens de marcação como HTML e estilização como CSS. Quando falamos do mundo Web a união entre HTML, CSS e JavaScript provê uma infinidades de recursos. Assim, nossos usuários poderão interagir em uma página não só bonita e animada, mas também com uma experiência incrível.

Apesar do JavaScript ser a linguagem de todos os navegadores existentes hoje, ele só se tornou popular após o framework para back-end NodeJS[1] ser criado. Assim, um desenvolvedor full-stack pode se aprofundar em apenas uma linguagem para programar todo o sistema. Outro ponto interessante é que a partir do NodeJS, outros frameworks foram desenvolvidos para possibilitar a programação com JavaScript em diversas frentes como:

- Electrom para programação de aplicativos Desktop;
- Johnny-Five para programação em IoT (Internet of Things) e robótica;
- MongoDB para trabalhar com bando de dados;
- E muitos outros.

Ao longo dessa apostila será desenvolvido de cadastro de consultas de uma nutricionista. A partir dele como nosso guia vamos abordar todos os conceitos existentes a partir da versão ES6 do JavaScript[2], que é a mais popular hoje em dia. Os arquivos iniciais do projeto podem ser encontrados no seguinte endereço https://github.com/amasiero/app-nutricao-js/tree/v1. Ao acessar o diretório e abrir o arquivo index.html no navegador, o resultado deve ser o mesmo apresentado pela Figura 1.

[1] https://nodejs.org/en/
[2] https://www.w3schools.com/js/js_es6.asp

*Figura 1. Página inicial do projeto*

# Manipulando o DOM

O usuário deseja que o sistema insira o seu nome antes do título “Nutricionista Esportiva”, assim ele pode ter uma experiência melhor garantindo que a lista de consultas apresentada é a sua lista e não de outro nutricionista.

O primeiro passo para atender essa necessidade do usuário é saber como pode-se inserir um script de JavaScript no arquivo HTML. Existem duas formas de fazer a inclusão de um script, a primeira é através da tag script, dentro do arquivo HTML:

```
<script type="text/javascript">
  // Codigo JavaScript
</script>
```

E a segunda maneira é através da inclusão de um arquivo JavaScript no HTML:

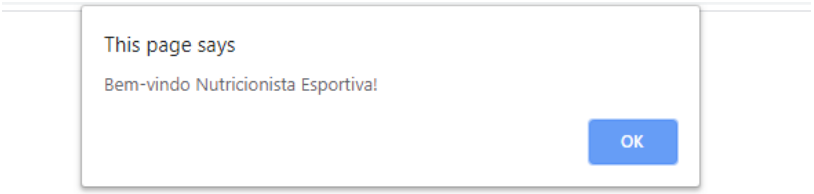
```
<script type="text/javascript" src="js/app.js"></script>
```

Agora que sabemos como incluir código JavaScript, vamos trabalhar com a exibição de informação para o usuário. A primeira maneira de exibir alguma informação para o usuário é através da função alert, ela apresenta uma caixa de confirmação com mensagem digitada, veja o código a seguir:

```
<script type="text/javascript">
  alert("Bem-vindo Nutricionista Esportiva!");
</script>
```

Agora confira o resultado na Figura 2.

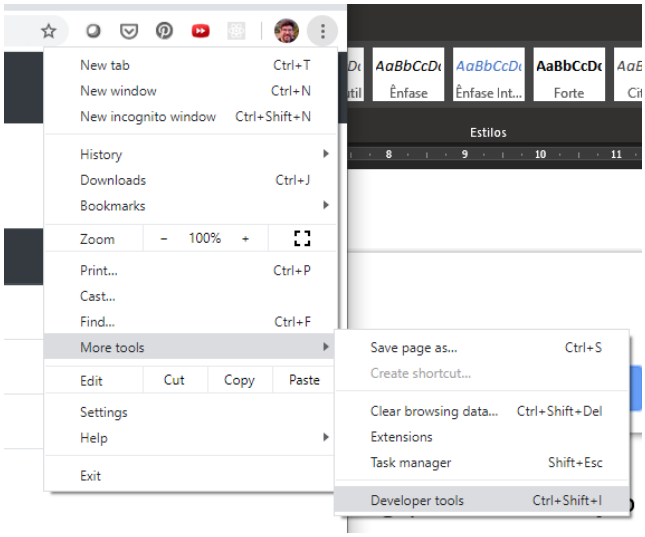
*Figura 2. Resultado da função alert escrita na página.*




A outra maneira é através da função console.log, porém essa função não exibe a informação diretamente para o usuário. Essa é uma função utilizada mais por desenvolvedores para acompanhar log de erro na aplicação do lado cliente. Para visualizar uma mensagem exibida através do console.log é necessário utilizar a ferramenta de desenvolvedor do seu navegador. A Figura 3 mostra o caminho de acesso a essa ferramenta no navegador Google Chrome, utilizado

durante os exercícios dessa apostila. Ela também pode ser acessada através da tecla de atalho Ctrl + Shift + I no Windows/Linux, ou Cmd + Shift + I no Mac.

*Figura 3. Acesso a ferramenta de desenvolvedor do navegador Google Chrome.*

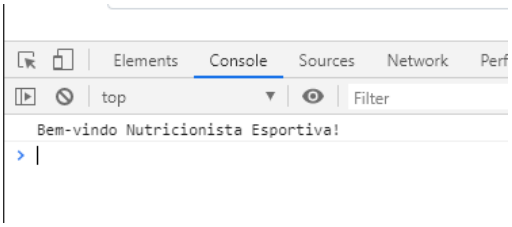



Vamos exibir a mesma mensagem do alert, agora com a função console.log e validar a saída através da ferramenta de desenvolvimento. O código alterado fica:

```
<script type="text/javascript">
  console.log("Bem-vindo Nutricionista Esportiva!");
</script>
```

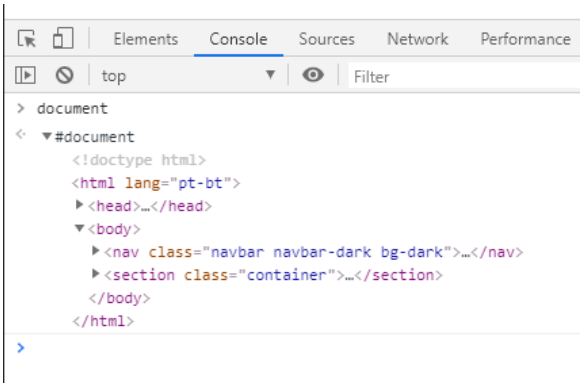
O resultado é apresentado através da

*Figura 4. Resultado apresentado através da função console.log, na ferramenta de desenvolvedor do Google Chrome.*




Tudo bem que agora podemos enviar informações do código em funções com a saída para o navegador, mas nenhuma delas resolve o problema do usuário. Para solucionar esse problema, é necessário manipular o arquivo HTML na sua essência. Aí, vem uma pergunta, como representar a página HTML dentro do JavaScript para conseguir então, para assim manipular seu conteúdo? Existe um objeto em JavaScript conhecido como DOM (Document Object Model). Ele contém todo o conteúdo de uma página HTML, com ele é possível acessar cada tag e suas propriedades. No código a palavra chave para ele é document, veja na Figura 5.

*Figura 5. Resultado do conteúdo armazenado no objeto document.*




Na Figura 5 é possível ver o resultado ao consultar o conteúdo do objeto document. Perceba que ele mantém todo o código HTML da página. Agora que sabemos como acessar o conteúdo HTML, é necessário percorre-lo para alterar a informação desejada. Para pesquisar alguma informação no document, o JavaScript disponibiliza uma função chamada querySelector (query => buscar, selector => seletor).

Com o querySelector é possível pesquisar qualquer elemento existente no DOM através dos seletores existentes no CSS:

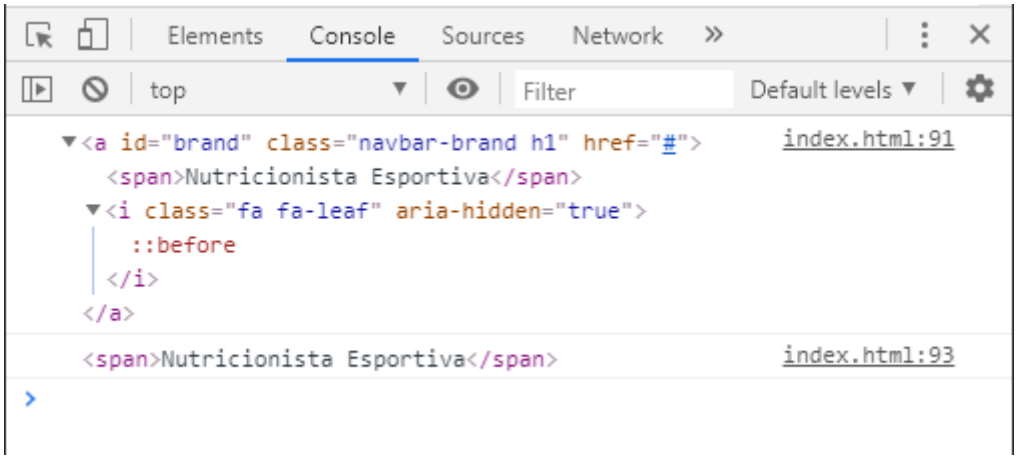
- O nome da tag;
- O nome da classe com um ponto final antes, ex: .classe;
- O nome do id da tag com uma hashtag antes, ex: #id.

Vamos então selecionar o elemento que representa o texto do título da página “Nutricionista Esportiva”. Veja o código para essa tarefa:

```
var titulo = document.querySelector("#brand");
console.log(titulo);
var textoTitulo = titulo.querySelector("span");
console.log(textoTitulo);
```

Perceba que no código foram separadas duas variáveis. A primeira titulo recebe o valor e uma busca pelo elemento com o id igual a brand. Dentro dele existe uma tag span que possui o texto do título, o qual devemos mudar para atender a necessidade do usuário. Mas antes veja o resultado obtido na Figura 6, perceba que titulo armazena o trecho do DOM correspondente ao id brand. A variável titulo também é um DOM, porém menor, assim ela também possui todas as propriedades e métodos do objeto anterior (document). Sendo assim, é possível realizar um novo querySelector agora para isolar a tag span, e o resultado é apresentado logo na sequência.

*Figura 6. Resultado obtido através do uso da função querySelector no DOM gerado.*




**Boa prática**

*Quando carregarmos o script da página, é importante que coloquemos ele como a última linha antes de fechar a tag body, pois assim garantirmos que o documento html está totalmente carregado.*

Agora que a propriedade desejada foi selecionada, como pode ser realizado a troca do conteúdo de texto dentro da tag? O DOM tem várias propriedades que auxiliam na manipulação do HTML. Uma delas é o textContent (Conteúdo de Texto). Com ela é possível manipular o conteúdo exibido para o usuário, o conteúdo que vai entre as tag do html. Veja o código atualizado.

```
var titulo = document.querySelector("#brand");
var textoTitulo = titulo.querySelector("span");
textoTitulo.textContent = "Luna Lovegood - Nutrição e Magia";
```

Na Figura 7, o resultado do novo HTML é apresentado.

*Figura 7. Resultado referente a alteração de conteúdo dinâmica através do DOM via JavaScript.*

**Boa prática**

*Atrelando a busca do querySelector a um nome de tag, podemos gerar um problema, pois se outra pessoa trocar a tag para melhorar o layout ou algo do tipo, a busca irá retornar como nula. Então é melhor utilizar outras informações que a função querySelector possibilita. Como ela faz a busca pelo seletor, pode-se utilizar seletores como no CSS para buscar elementos como .classe e o #id. Use sempre algum seletor que irá representar melhor a informação que você deseja buscar no objeto HTML.*

Com o objetivo do usuário atingido, devemos criar um arquivo para manter as responsabilidade de cada linguagem no arquivo correspondente. Dessa forma, a manutenção do código fica mais fácil.

**Boa prática**

*É importante separar os conteúdos das ferramentas em seus devidos e apropriados locais. Isso quer dizer que o código HTML fica no arquivo HTML, o código CSS no arquivo CSS e o código JavaScript no arquivo JavaScript. Depois de criar o arquivo, basta vincular ele à página HTML, como exemplo:*

```
<script type="text/javascript" src="js/app.js"></script>
```

Nossa primeira tarefa foi concluída com sucesso. Estude mais e refaça os passos para fixar melhor o conteúdo. Tente refazer sem consultar o material.

## O que aprendemos

- Como inserir um script na página HTML;
- Funções alert e console.log para exibir saídas do script;
- Função querySelector para selecionar elementos do DOM;
- Propriedade textContent para mudar o conteúdo de texto de uma determinada tag;
- Manter o script como a última tag antes de fechar a tag body;
- Separar os arquivos de acordo com suas responsabilidades.

O código atualizado encontra-se em https://github.com/amasiero/app-nutricao-js/tree/v2

# Recuperando informações do formulário

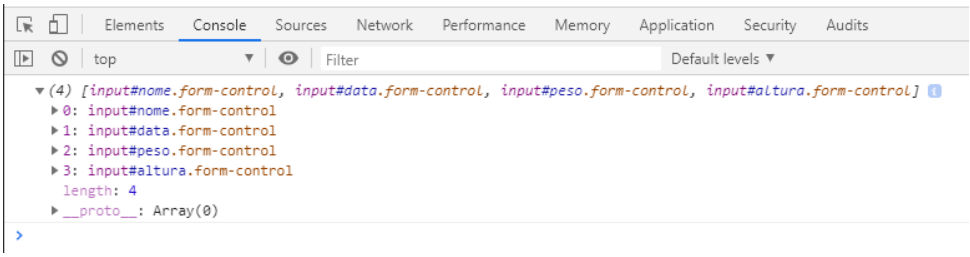
O usuário deseja preencher os dados da consulta e após clicar no botão “Confirmar”, os dados devem ser inseridos na tabela de consultas.

Se observarmos o código HTML do formulário, cada input possui um identificador único. Esse identificador é o valor da propriedade id, existente em cada uma dos 4 campos de entrada de informação. Com essa informação em mãos e o conhecimento da função querySelector, podemos atribuir o objeto DOM de cada campo a um array. Veja o código:

```
var campos  = [
    document.querySelector('#nome'),
    document.querySelector('#data'),
    document.querySelector('#peso'),
    document.querySelector('#altura')
];

console.log(campos);
```

Criamos uma variável campos, que armazena o DOM de cada um dos campos, assim fica mais fácil manipular as informações. A Figura 8 apresenta a saída com os DOMs armazenados na variável campos.

*Figura 8. Resultado da lista de DOMs gerada a partir da função querySelector e atribuição a variável campos.*




A partir desse momento, para acessar quaisquer valores digitados no formulário, basta dizer o índice do campo que deseja recuperar o valor. A ordem da lista é nome, data, peso e altura. Nesse momento, é preciso informar que o formulário deve ser enviado/submetido quando o botão “Confirmar” for clicado. Durante o processo de submissão do formulário, as informações digitadas nos inputs serão capturadas e as linhas (tr) da tabela serão montadas dinamicamente.

Com a função document.querySelector, a tag form é selecionada. Nela adicionaremos um evento do tipo submit, que está relacionado ao envio, à submissão do formulário. Para essa tarefa a função addEventListener executa esse papel, para disparar uma função quando o formulário for submetido. Essa

função executada é o que chamamos de função de callback, que é disparada ao clicar no botão “Confirmar”.

A primeira forma de trabalhar com a função de callback é declarar uma função e chama-la na função addEventListener, da seguinte maneira:

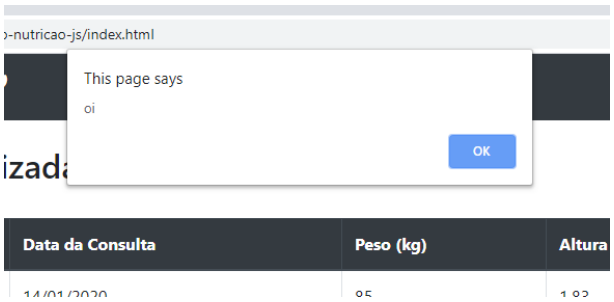
```
function callback(evento) {
    alert('oi');
}

document.querySelector('form').addEventListener('submit', callback);
```

Entretanto, o JavaScript possibilita o trabalho com funções chamadas anônimas. Essas funções são declaradas sem a necessidade de definir um nome para elas. Assim, o novo código deve ficar da seguinte maneira:

```
document.querySelector('form').addEventListener('submit',
function(evento) {
    alert('oi');
});
```

Vamos conferir o resultado da execução do evento submit através da Figura 9.

*Figura 9. Resultado da chamada da função anônima após o clique do botão Confirmar.*




A partir do momento que as informações dos inputs são capturadas e o evento é disparado pelo clique do botão “Confirmar”, deve-se criar uma tag tr para inserir as informações na tabela. Para a criação de uma tag, o DOM possui uma função chamada createElement, que recebe como parâmetro o nome da tag desejada.

```
var tr = document.createElement('tr');
```

Cada tr, precisa das td´s que recebem os valores respectivos de cada consulta. Como temos a variável campos, que é uma lista ou array, ela possui uma função chamada forEach. Essa função percorre cada elemento da lista e executa uma determinada ação. Essa ação é feita através de uma função

anônima que recebe como parâmetro o campo referente a iteração atual do for. A primeira vez que executar, o parâmetro recebe o input referente ao id nome. Na sequência o id data, e assim por diante. Dentro da função executada pelo forEach, será acessado o valor digitado no input através da propriedade value e esse valor deve ser atribuído a propriedade textContent da td criada dinamicamente. Confira o código:

```
campos.forEach(function(campo) {
   var td = document.createElement('td');
   td.textContent = campo.value;
});
```

Depois de criada a td e seu conteúdo de texto foi atribuído corretamente, ele deve ser inserido dentro da tag tr, afinal td é filha de tr. Esse vínculo deve ser realizado através da função appendChild. Confira:

```
campos.forEach(function(campo) {
   var td = document.createElement('td');
   td.textContent = campo.value;
   tr.appendChild(td);
});
```

Agora ao chegar no fim da execução do forEach, é necessário calcular o valor do IMC do paciente. A equação utilizada é:

$$IMC = \frac{peso}{altura^2}$$

O código do cálculo e da criação da tag td é apresentado na sequência:

```
var td = document.createElement('td');
td.textContent = (
    campos[2].value /
    (campos[3].value * campos[3].value)
).toFixed(2);
tr.appendChild(td);
```

A função toFixed determina o número máximo de casas decimais na exibição dele para o usuário. Vamos adicionar agora a linha da tabela, selecionando a tbody da table através do querySelector. Por uma questão de desempenho, vamos realizar a chamada do querySelector para a tabela fora da função de clique, assim evitamos que o código percorra o documento HTML inteiro para selecionar a tabela, a cada clique. O código completo ficou assim:

```
var tbody = document.querySelector('table tbody');

document.querySelector('form').addEventListener('submit',
```

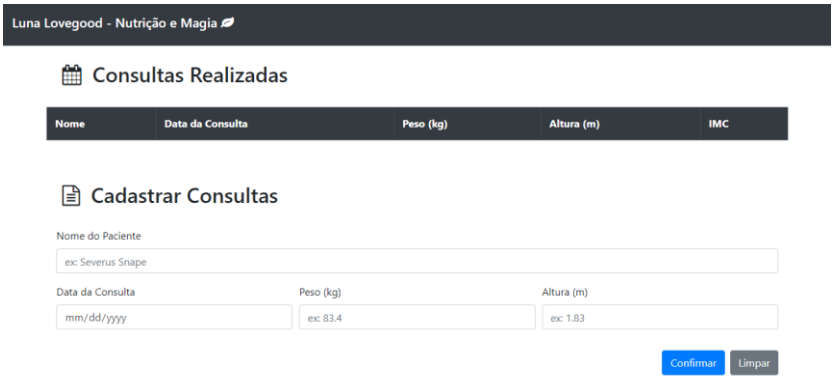
```
function(evento) {
    var tr = document.createElement('tr');

    campos.forEach(function(campo) {
       var td = document.createElement('td');
       td.textContent = campo.value;
       tr.appendChild(td);
    });

    var td = document.createElement('td');
    td.textContent = (
        campos[2].value /
        (campos[3].value * campos[3].value)
    ).toFixed(2);
    tr.appendChild(td);

    tbody.appendChild(tr);

});
```

Vamos remover as linhas da tabela para zerar e testar para ver se nosso código está funcionando de acordo.

```
<table class="table table-bordered">
  <thead class="thead-dark">
    <tr>
      <th scope="col">Nome</th>
      <th scope="col">Data da Consulta</th>
      <th scope="col">Peso (kg)</th>
      <th scope="col">Altura (m)</th>
      <th scope="col">IMC</th>
    </tr>
  </thead>
  <tbody>

  </tbody>
</table>
```

Preenchendo o formulário e clicando no botão percebemos que o resultado é o mesmo apresentado na Figura 10.

*Figura 10. Resultado ao inserir os dados do formulário.*



Ao clicar no botão é notável que a tela é atualizada no navegador, mas a consulta não aparece na tabela. Isso ocorre, pois, o evento de submit envia uma pedido (request) ao servidor. Na sequência o servidor devolve uma resposta (response), e como não foi informada a ação que o servidor deveria fazer, apenas a página é recarregada com seu estado original. Não estamos trabalhando com banco de dados, então a informação inserida é simplesmente removida.

Para evitar esse tipo de situação, utilizamos o parâmetro evento, e chamamos a função preventDefault. Ela evitará o processo de submissão e consequentemente a página não será carregada. Veja o código:

```
document.querySelector('form').addEventListener('submit',
function(evento) {
    evento.preventDefault();
    var tr = document.createElement('tr');

    // Código ocultado…

});
```

Agora, vamos conferir o resultado através da Figura 11.

*Figura 11. Consulta inserida na tabela com sucesso.*


Pronto, agora a consulta foi inserida com sucesso. Não se preocupe com o formato da data, iremos acertar ele em breve. Perceba que possuímos um pequeno problema no nosso formulário, ele não apagou os dados inseridos. Para uma experiência do usuário isso é ruim, afinal ele terá que apagar todos os dados para inserir uma nova consulta. Então, após inserir a linha na tabela, vamos limpar o formulário e colocar o foco novamente no nome do paciente.

```
document.querySelector('form').addEventListener('submit',
function(evento) {

    // Código ocultado…
    tbody.appendChild(tr);

    this.reset();
    campos[0].focus();

});
```

Como o evento foi associado ao formulário, utilizamos a palavra this para nos referirmos ao próprio formulário. Como ele acessamos uma função reset, que leva nosso formulário ao estado original, em branco. Na sequência, acessamos através da lista campos o objeto que representa o input do nome do paciente, e chamamos a função focus para retornar o foco das entradas (mouse e teclado) para esse input. O resultado é apresentado na Figura 12.

*Figura 12. Formulário limpo e foco no nome do paciente.*


Conseguimos inserir os dados da consulta de acordo com a entrada do usuário. Porém, alguns pontos ainda precisam ser melhorados. O formato de exibição da data não está com o formato comumente adotado no Brasil, e também não definimos o que representa efetivamente uma consulta. As regras de negócio e manipulação de interface gráfica do usuário devem ser separadas em camadas diferentes, assim a manutenção e organização do código será mais eficiente. A arquitetura proposta utilizada para suprir essa necessidade é a arquitetura em camadas MVC (Model-View-Control).

Além disso, vamos entrar no paradigma de programação Orientada à Objetos. Assim estruturaremos melhor o nosso código. Mas, por enquanto, ficamos com nosso sistema assim.

## O que aprendemos

- Criação de uma array com os campos do formulário;
- A função addEventListener para trabalhar com eventos em elementos HTML;
- Criação de função;
- Criação de uma função anônima;
- Função toFixed para exibir a quantidade de casas decimais desejadas;
- Criar um elemento HTML através da função createElement;
- Incluir um elemento HTML através da função appendChild;
- Trabalhar com o forEach;
- A função preventDefault para evitar que a página seja recarregada;
- Palavra reservada this;
- Função reset para limpar o formulário;
- Função focus para direcionar o cursor ao elemento desejado.

O código atualizado encontra-se em https://github.com/amasiero/app-nutricao-js/tree/v3

# Especificando o negócio

Como discutido no capítulo anterior, o que é uma consulta não foi definido muito bem. Para atender melhor o nosso usuário, a arquitetura do código deve refletir esses detalhes de modelagem. Uma boa arquitetura e modelagem pode melhorar não só o desenvolvimento, mas também o desempenho do código. Assim, o primeiro passo para atender nossa estrutura é definir o modelo de uma consulta.

## O que é um modelo?

Um modelo nada mais é do que a abstração do mundo real. Por exemplo, quando a defesa civil prepara seu atendimento em situações de desastres naturais, criando um modelo do local estudado. Nesse modelo ela insere todas as entradas ou informações necessárias, com quantidade de chuva, velocidade do vento e descrevem o passo a passo do que pode acontecer quando esses fatores atingem valores alarmantes. Em nosso caso, iremos criar um modelo para nossa consulta.

Dado o objetivo do sistema que é cadastrar as consultas do paciente de uma nutricionista, define-se quais são os atributos adequados para a representação de uma consulta no sistema. Esse modelo, deve ser capaz de fazer de maneira programática tudo que seria feito na vida real. A criação de um modelo em JavaScript necessita do uso de um paradigma de programação muito utilizado, que é a orientação à objetos. De maneira resumida é a criação de uma classe, que representa uma receita de bolo, ou todos os ingredientes que o bolo poderá ter ao criar um objeto dele. Para nossa sorte, a versão ES6 do Javascript facilitou a criação de classes, o que no passado não era tão trivial assim.

## Organizando o sistema

Sabendo disso, o primeiro passo é a criação de uma classe chamada Consulta facilitando a comunicação dentro do sistema. Para construirmos nosso sistema, vamos adotar uma convenção de organização dos arquivos. Ela auxiliará na aplicação da arquitetura em camadas MVC.

Dentro da pasta js, criaremos uma nova pasta chamada app. Em app cria-se uma subpasta chamada models. A pasta models abrigará todos os modelos do sistema. Nela criamos o arquivo Consulta.js, que terá todo o código da classe Consulta. Ela adota o padrão de nomenclatura CamelCase[3] e para classes a

[3] https://www.w3schools.com/js/js_conventions.asp

primeira letra deve ser em caixa alta (letra maiúscula). É um padrão pouco comum em JavaScript, mas deixa claro que o arquivo se trata de uma classe. Veja a implementação da classe Consulta:

```
class Consulta {
    constructor() {
        this.nome = '';
        this.data = new Date();
        this.peso = 0.0;
        this.altura = 0.0;
    }
}
```

No ES6, para criar a classe utiliza-se a palavra reservada class seguida pelo nome da classe. Por convenção segue o mesmo nome dado ao arquivo, no exemplo, Consulta. É apenas por uma questão de organização, diferente do Java, o nome do arquivo não interfere no nome da classe. Para definir os atributos da classe é utilizado a função constructor.

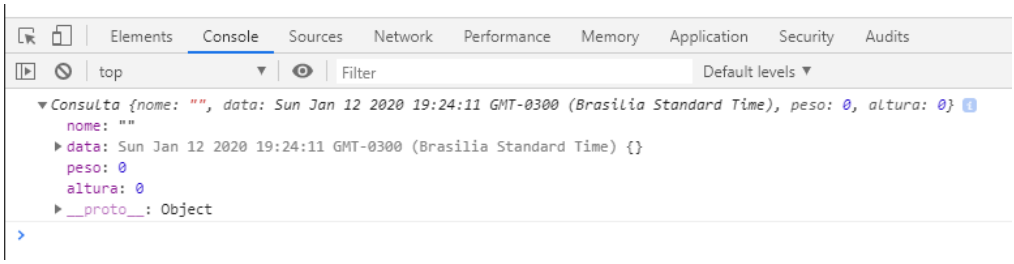
Nesse momento, sabemos que toda consulta possui o nome do paciente, data, peso e altura do paciente. O **this** é uma variável implícita que sempre apontará para a instância do objeto que está executando a operação no momento.

Agora é preciso adicionar o arquivo no HTML e vamos aproveitar para criar um objeto de Consulta. No arquivo index.html:

```
<script type="text/javascript"
src="js/app/models/Consulta.js"></script>
<script type="text/javascript">
  var consulta = new Consulta();
  console.log(consulta);
</script>
```

Veja o resultado do código no console do Chrome, de acordo com a Figura 13.

*Figura 13. Instância do objeto Consulta apresentada no console.*




A palavra new é utilizada para que o interpretador saiba que um novo objeto será alocado na memória. Ela invoca o método constructor para determinar os

espaços dos atributos da classe, nesse caso nome, data, peso e altura. Todos inicializados com seus valores mínimos determinados no código.

**Curiosidade**

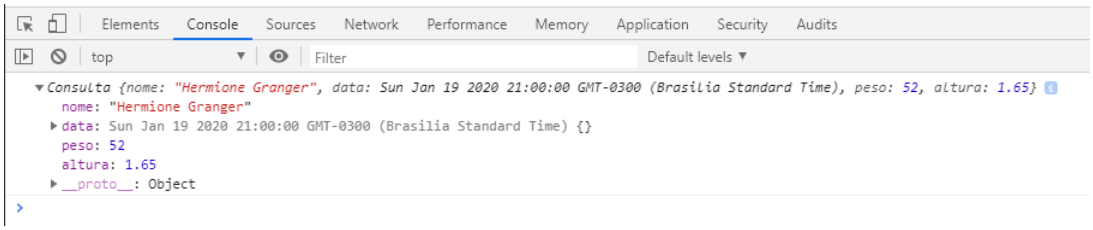
*Quando uma função está associada a uma classe, chamamos de método. Por exemplo, o método constructor.*

E como podemos atribuir uma valor para os atributos da classe Consulta. Veja o código a seguir.

```
var consulta = new Consulta();
consulta.nome = 'Hermione Granger';
consulta.data = new Date('2020-01-20');
consulta.peso = 52;
consulta.altura = 1.65;
console.log(consulta);
```

Basta usar o nome do objeto seguido de um ponto e o nome do atributo que você deseja atribuir o valor. Confira o resultado através da Figura 14.

*Figura 14. Consulta com os dados preenchidos.*




## Melhorando o construtor e os métodos da classe

Se analisarmos o projeto, uma consulta existe no sistema apenas quando todos os atributos são informados. Dessa maneira, podemos gerar uma melhora no código da classe. Nesse caso, forçamos o construtor a receber como parâmetro valores referentes a todos os atributos. Confira o código:

```
class Consulta {
    constructor(nome, data, peso, altura) {
        this.nome = nome;
        this.data = data;
        this.peso = peso;
        this.altura = altura;
    }
}
```

E a instância passa a ser gerada da seguinte maneira:

```
var consulta = new Consulta(
        'Hermione Granger',
        new Date('2020-01-20'),
        52, 1.65 );
```

O próximo ponto é o cálculo do IMC, afinal ele não está no formulário, mas deve ser calculado de maneira automática. Nesse caso, a melhor opção é a criação de um método. Assim, todas as informações da consulta podem ser obtidas através do modelo. Confira a classe Consulta final:

```
class Consulta {
    constructor(nome, data, peso, altura) {
        this.nome = nome;
        this.data = data;
        this.peso = peso;
        this.altura = altura;
    }

    calculaIMC() {
        return this.peso / (this.altura * this.altura);
    }
}
```

## Encapsulamento

Existe uma regra dada pela nutricionista, que a consulta é única. Ela não pode ser alterada. A implementação da classe Consulta, permite realizar essa alteração. Então, como cumprir com esse requisito funcional? Existe uma convenção em JavaScript para deixar os atributos somente como leitura. Na verdade, é o princípio de encapsulamento da orientação à objetos.

Contudo, não existe até o momento, um modificador de acesso em JavaScript. A convenção que utilizamos é colocar um underline (_) antes do nome do atributo para informar que não pode ser modificado. Veja a implementação:

```
class Consulta {
    constructor(nome, data, peso, altura) {
        this._nome = nome;
        this._data = data;
        this._peso = peso;
        this._altura = altura;
    }

    calculaIMC() {
        return this._peso / (this._altura * this._altura);
```

```
    }
}
```

Essa convenção apenas informa ao desenvolvedor que as propriedades que contenham o underline só poderão ser acessadas através dos métodos de acesso (*getters* e *setters*). Os métodos são:

```
calculaIMC() {
    return this._peso / (this._altura * this._altura);
}

getNome() {
    return this._nome;
}

getData() {
    return this._data;
}

getPeso() {
    return this._peso;
}

getAltura() {
    return this._altura;
}
```

Para manter o padrão de nomenclatura, altere o nome do método calculaIMC para getIMC:

```
getIMC() {
    return this._peso / (this._altura * this._altura);
}
```

O JavaScript nos permite deixar o código para os métodos getters mais limpo. Ele possui uma propriedade get que facilita a implementação e depois a chamada as propriedades no código. Confira:

```
get imc() {
    return this._peso / (this._altura * this._altura);
}

get nome() {
    return this._nome;
}

get data() {
    return this._data;
}

get peso() {
    return this._peso;
```

```
}

get altura() {
    return this._altura;
}
```

A chamada no HTML fica da seguinte maneira agora:

```
var consulta = new Consulta(
        'Hermione Granger',
        new Date('2020-01-20'),
        52, 1.65 );
console.log(consulta.imc); // Ao invés de consulta.getIMC()
```

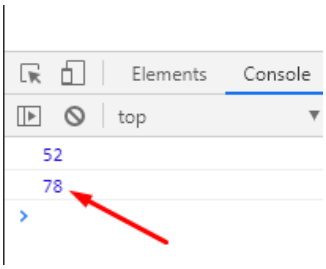
## Está mesmo imutável agora?

Vamos testar o seguinte código:

```
var consulta = new Consulta(
        'Hermione Granger',
        new Date('2020-01-20'),
        52, 1.65 );
consulta.peso = 60;
console.log(consulta.peso);
consulta._peso = 78;
console.log(consulta.peso);
```

Confira o resultado na Figura 15:

*Figura 15. Alteração do valor do atributo privado.*



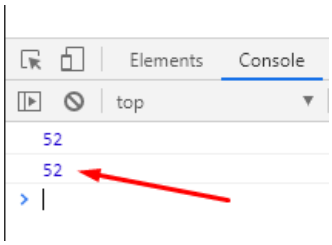

Ao utilizar o atributo com o underline o valor foi alterado de qualquer maneira. Isso não poderia acontecer, correto? Sim, não poderia. Para corrigir esse problema, devemos congelar a instância do objeto, meio que deixá-lo como uma constante. Assim, qualquer tentativa de alteração das propriedades será ignorada pelo sistema. O método Object.freeze auxiliará nesse propósito:

```
class Consulta {
    constructor(nome, data, peso, altura) {
        this._nome = nome;
        this._data = data;
        this._peso = peso;
        this._altura = altura;
```

```
        Object.freeze(this);
    }

    // Restante do código omitido...

}
```

Vamos ver o resultado. Confira na Figura 16:

*Figura 16. Resultado após congelar o objeto.*

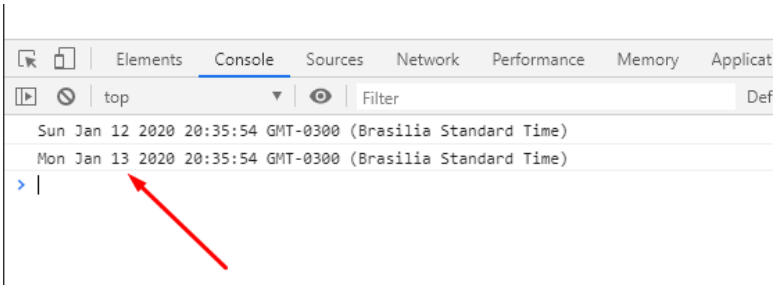



Mas será que está tudo imutável agora? Vamos conferir o cenário a seguir:

```
var consulta = new Consulta(
        'Hermione Granger',
        new Date(),
        52, 1.65 );
console.log(consulta.data);
consulta.data.setDate(13);
console.log(consulta.data);
```

Agora vamos conferir o resultado através da Figura 17.

*Figura 17. Tentativa de alteração na data feita com sucesso.*




A data foi modificada, mas isso não deveria ter acontecido! O Object.freeze é o que chamamos de objeto shallow (não tem nada a ver com a Lady Gaga). Isso significa que é um objeto de superfície, ele congela valores e não instâncias de outros objetos que temos nos atributos como o caso do _data, um objeto do tipo Date.

Uma maneira de resolver isso é utilizando uma técnica de programação defensiva para essa situação. Para blindar o atributo, no método get retornamos uma nova instância do objeto _data:

```
get data() {
    return new Date(this._data.getTime());
}
```

O construtor da classe Date recebe o valor do método getTime que é um valor long representando a data. Assim, a instância retornada no objeto.data é outra diferente da armazenada. Dessa forma, conseguimos blindar nosso atributo. A técnica de programação defensiva também deve ser aplicada no construtor da classe:

```
constructor(nome, data, peso, altura) {
    this._nome = nome;
    this._data = new Date(data.getTime());
    this._peso = peso;
    this._altura = altura;
    Object.freeze(this);
}
```

Quando construir a classe, é importante fazer com que ela seja blindada de acordo com as regras de negócio. Fique sempre atento e utilize a programação defensiva sempre que necessário.

**Boa Prática**

*Ao invés de utilizar a palavra **var** na declaração de objetos, utilize a palavra **let**. A **var** é utilizada em acesso global, não tem escopo de bloco. Já **let** é o tipo de declaração que mantém o escopo de bloco, o que é mais recomendado.*

## O que aprendemos

- O que é um modelo
- Classe
- Construtor da classe
- Método de classe
- Encapsulamento
- Propriedade get
- Object.freeze
- Programação Defensiva

O código atualizado encontra-se em https://github.com/amasiero/app-nutricao-js/tree/v4

# Pegando o controle da aplicação

Com o modelo da consulta criado, é preciso capturar as ações do usuário e interagir com o modelo. Na arquitetura MVC o responsável por fazer essa ligação é a camada de controle. Para isso, vamos criar a classe ConsultaController.js na subpasta controllers ao lado da subpasta models. O código da classe fica assim:

```
class ConsultaController {
    adiciona(evento) {
        evento.preventDefault();
        alert('Ação executada');
    }
}
```

O método adiciona será chamado quando o usuário clicar no botão Confirmar. Se não utilizarmos o evento.preventDefault(), a página será recarregada e desejamos cancelar esse comportamento. Vamos agora, vincular a classe no HTML e criar um objeto controller:

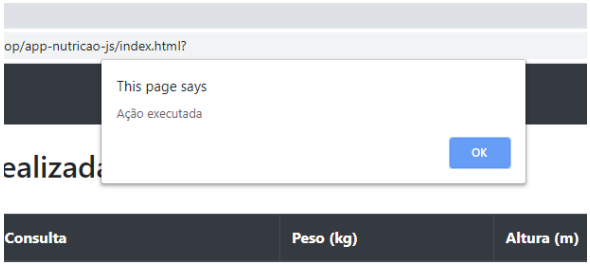
```
<script type="text/javascript"
src="js/app/models/Consulta.js"></script>
<script type="text/javascript"
src="js/app/controllers/ConsultaController.js"></script>
<script type="text/javascript">
  let consultaController = new ConsultaController();
</script>
```

O objeto é criado para utilizarmos na página HTML. Agora na tag form, vincula-se a propriedade onsubmit com o método adiciona da classe ConsultaController:

```
// Código omitido…
</h2>
<form class="col-12" onsubmit="consultaController.adiciona(event)">
  <div class="form-group">
// Código omitido…
```

O resultado do teste do método adiciona é apresentado na Figura 18.

*Figura 18. Resultado do vínculo do método adiciona com evento de submit do formulário.*

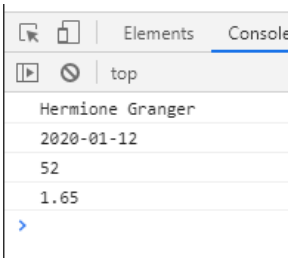


O próximo passo é capturar os campos do formulário para poder construir o objeto de consulta.

```
adiciona(evento) {
    evento.preventDefault();
    var inputNome = document.querySelector('#nome');
    var inputData = document.querySelector('#data');
    var inputPeso = document.querySelector('#peso');
    var inputAltura = document.querySelector('#altura');

    console.log(inputNome.value);
    console.log(inputData.value);
    console.log(inputPeso.value);
    console.log(inputAltura.value);
}
```

Confira o resultado na Figura 19.

*Figura 19. Resultado da captura dos valores digitados no formulário.*

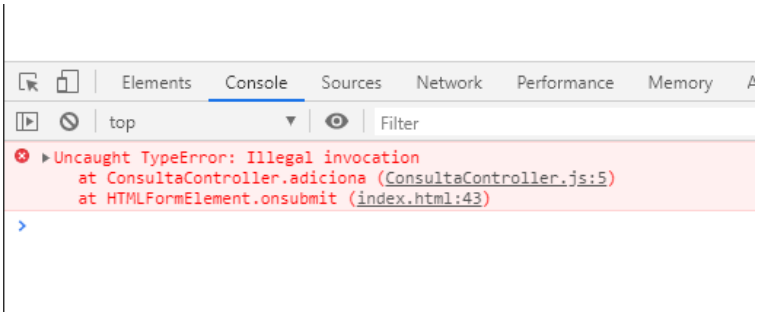



Perceba que há muita repetição no código. A sintaxe ficou trabalhosa com necessidade de muita digitação. Em JavaScript temos as First Class Functions, onde podemos declarar a variável $ (como no jQuery) e atribuirmos a função document.querySelector. O código fica assim:

```
var $ = document.querySelector;
var inputNome = $('#nome');
var inputData = $('#data');
var inputPeso = $('#peso');
var inputAltura = $('#altura');
```

Veja o que acontece agora quando preenchemos o formulário na Figura 20.

*Figura 20. Erro ao trabalhar com First Class Function para querySelector.*



Não funcionou a atribuição do querySelector na variável $, criando assim um alias. Mas o que aconteceu? O querySelector é um método que pertence ao objeto document. Internamente, o querySelector tem uma chamada para o this, que é o contexto pelo qual o método é chamado. Sendo assim, o this é referência à instância de document. No entanto, quando colocamos o querySelector dentro do $, ele passa a ser executado fora do contexto de document, logo não funciona. É necessário tratar o querySelector como uma função separada. A ideia é que ao atribuirmos o querySelector para o $, ele mantenha sua associação com o document. Para isto, usaremos o método bind():

```
var $ = document.querySelector.bind(document);
```

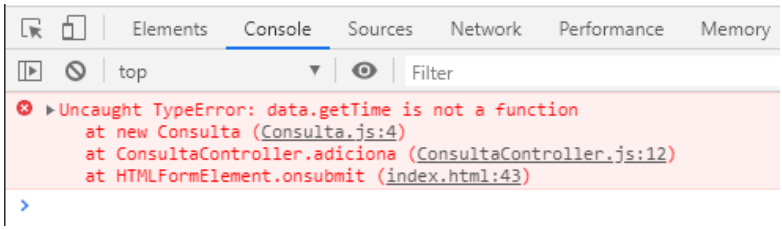
Assim informamos que o querySelector é atribuído à variável $, mas permanece associado com document. Esse é um truque similar ao do jQuery. Ele cria um "mini-framework", quando há a associação da variável $ com o querySelector e mantendo a ligação com o document.

O que acontece agora é que percorremos o document toda vez que vamos realizar o cadastro de uma nova consulta. Para minimizarmos essa chamada e deixarmos o código com um desempenho melhor, utilizaremos um construtor na classe controller. Já aproveitando, vamos também instanciar um objeto consulta com os valores recebidos através dos inputs do formulário:

```
class ConsultaController {
    constructor() {
        let $ = document.querySelector.bind(document);
        this._inputNome = $('#nome');
        this._inputData = $('#data');
        this._inputPeso = $('#peso');
        this._inputAltura = $('#altura');
    }

    adiciona(evento) {
        evento.preventDefault();
        let consulta = new Consulta(
            this._inputNome.value,
            this._inputData.value,
            this._inputPeso.value,
            this._inputAltura.value
        );

        console.log(consulta);
    }
}
```

Vamos conferir o resultado da invocação do método adiciona através da Figura 21.

*Figura 21. Erro ao criar uma instância de Consulta.*

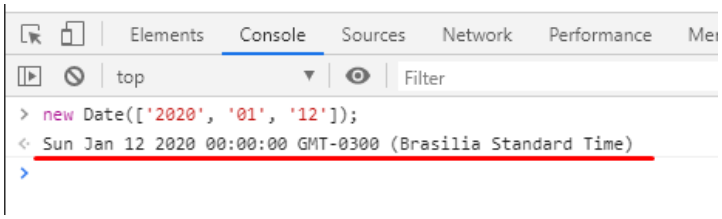



O problema é que o valor que estamos passando para criação do objeto Date é do tipo string, que é o tipo de todos os valores recuperados dos inputs. Então, como podemos resolver isso?

## Criando um objeto Date

O objeto Date possui um construtor que recebe um array de strings onde ele espera os valores de ano, mês e dia. Confira na Figura 22.

*Figura 22. Resultado do construtor da classe Date para um array de strings.*




Uma maneira de resolver isso é utilizar a função split:

```
let consulta = new Consulta(
    this._inputNome.value,
    new Date(this._inputData.value.split('-')),
    this._inputPeso.value,
    this._inputAltura.value
);
```

Porém, um dos construtores existentes na classe Date recebe uma string no seguinte formato “ano, mês, dia”. Sendo assim, podemos utilizar a função replace e aplicar uma expressão regular para substituir o – por , na string da data recebida pelo input.

```
new Date(this._inputData.value.replace(/-/g, ','))
```

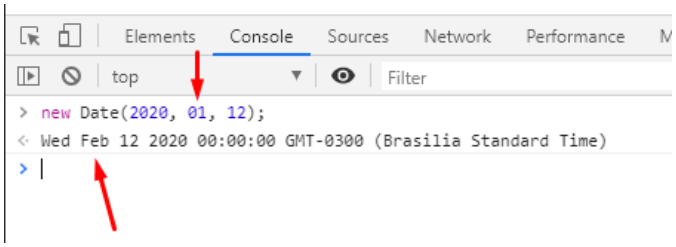
## O problema com Datas

Existe outra maneira de criar um objeto do tipo Date. Veja a linha de código a seguir:

```
let data = new Date(2020, 01, 12);
```

A Figura 23 apresenta o resultado obtido com essa linha de código.

*Figura 23. Resultado da criação de um objeto Date com 3 parâmetros.*




Informamos a data com o mês de janeiro, mas foi exibido o mês de fevereiro. Nesse formato o mês deve ser informado dentro de um intervalo entre 0 e 11, inclusive. Então, toda vez que iremos realizar a criação do objeto utilizando esse tipo de construtor, devemos subtrair um do mês.

Vamos utilizar o paradigma funcional para solucionar esse problema de criar o objeto Date. No ES6 existe o recurso do spread operator, veja:

```
new Date(...this._inputData.value.split('-'))
```

Adicionamos ... (reticências) posicionado antes do this. O spread operator informa que o primeiro item do array (gerado com o método split) ocupará o primeiro parâmetro do construtor, e assim cada parâmetro do Date será posicionado na mesma ordem no construtor. A data será passada, mas o mês ficará incorreto.

Para resolver, vamos trabalhar com a função map(), bem conhecida no mundo JavaScript. Ela nos permitirá subtrair 1 do mês. Então, iremos chamar a função map no array criado e dependendo do elemento, iremos diminuir -1. Essa é a estrutura inicial do uso da função map.

```
new Date(...
    this._inputData.value
        .split('-')
        .map(function(item) {
            return item;
        })),
```

Mas até esse momento o problema da data não foi resolvido, pois só está retornando o item recebido. Um segundo parâmetro na função map() pode ser adicionado, é o índice. Incluiremos um if, no qual quando o índice for igual a 1, vamos subtrair 1.

```
new Date(...
    this._inputData.value
        .split('-')
```

```
        .map(function(item, indice) {
            if(indice == 1) {
                return item -1;
            }
            return item;
        })),
```

Assim, nosso problema foi resolvido. Contudo, existe uma maneira mais elegante de resolver isso. São três parâmetros e só gostaria de subtrair um do parâmetro de índice ímpar. Sendo assim, podemos utilizar o operador módulo para efetuar a conta:

```
new Date(...
    this._inputData.value
        .split('-')
        .map(function(item, indice) {
            return item - (indice % 2);
        })),
```

Existe uma maneira de deixar o código ainda menos verboso. A função anônima dentro de map, pode ser escrita como uma Arrow Function. As arrow functions possuem uma sintaxe curta ao compara-las com as funções tradicionais, porém não possuem this, arguments, super ou new.target. Elas devem ser aplicadas para funções que não sejam métodos de classes. Também não podem ser utilizadas como construtores.

```
map((item, indice) => {
    return item - (indice % 2);
})
```

Como existe apenas uma linha de código com o return, a expressão pode ficar ainda menos verbosa. Veja:

```
map((item, indice) =>  item - (indice % 2))
```

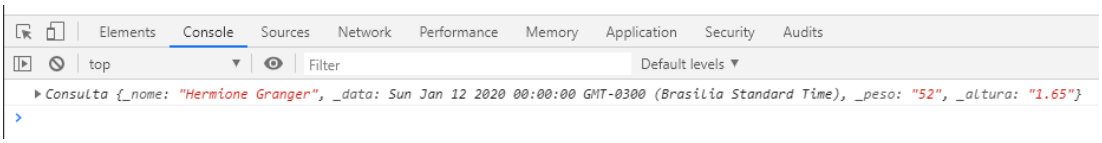
Confira o resultado da classe completa agora:

```
class ConsultaController {
    constructor() {
        let $ = document.querySelector.bind(document);
        this._inputNome = $('#nome');
        this._inputData = $('#data');
        this._inputPeso = $('#peso');
        this._inputAltura = $('#altura');
    }

    adiciona(evento) {
        evento.preventDefault();
        let consulta = new Consulta(
```

```
            this._inputNome.value,
            new Date(...
                this._inputData.value
                    .split('-')
                    .map((item, indice) =>  item - (indice % 2))
            ),
            this._inputPeso.value,
            this._inputAltura.value
        );

        console.log(consulta);
    }
}
```

A Figura 24 apresenta agora o resultado da criação do objeto consulta agora:

*Figura 24. Consulta criada com sucesso a partir do formulário.*




## O que aprendemos

- Camada de controller
- First Class Functions
- Método bind
- A ideia por trás de um mini-framework
- Construtores Date
- Spread operator
- Map Function
- Arrow Function

O código atualizado encontra-se em https://github.com/amasiero/app-nutricao-js/tree/v5

## Isolando as responsabilidades

Antes de colocar a linha com as informações da consulta na tabela é necessário fazer com que a exibição da data esteja no formato mais apropriado para o usuário. O ideal é utilizar o padrão brasileiro que é dia / mês / ano (com 4 dígitos). Mas perceba que estamos atribuindo responsabilidades além do controller para a classe ConsultaController. Além disso, essas operações com data poderão ser utilizadas em outras partes do sistema também. Isso está cheirando a código repetido! Nesse caso, o ideal é construir uma classe que irá auxiliar o sistema inteiro com problemas referentes a manipulação de data.

Classes que prestam um serviço de apoio em nosso sistema são chamadas de helpers (ajudantes ou utilitários). Vamos criar uma classe chamada DateHelper que deverá estar dentro da subpasta helpers. Nela existirão dois métodos um que irá receber uma data e irá converter para texto e outra que irá receber o texto para converter para data. Confira o código:

```
class DateHelper {
    textoParaData(texto) {
        return new Date(...
            texto
                .split('-')
                .map((item, indice) => item - (indice % 2)));
    }

    dataParaTexto(data) {
        return data.getDate()
            + "/" + (data.getMonth() + 1)
            + "/" + data.getFullYear();
    }
}
```

Vamos incluir agora  chamada da classe no controller:

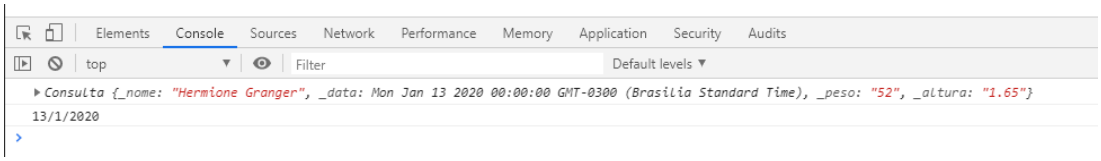
```
let helper = new DateHelper();
let consulta = new Consulta(
    this._inputNome.value,
    helper.textoParaData(this._inputData.value),
    this._inputPeso.value,
    this._inputAltura.value
);

console.log(helper.dataParaTexto(consulta.data));
```

Por fim, devemos incluir o script no index.html e realizar o teste:

```
<script type="text/javascript"
src="js/app/models/Consulta.js"></script>
<script type="text/javascript"
src="js/app/helpers/DateHelper.js"></script>
<script type="text/javascript"
src="js/app/controllers/ConsultaController.js"></script>
<script type="text/javascript">
  let consultaController = new ConsultaController();
</script>
```

A Figura 25 apresenta o resultado obtido através do uso do helper.

*Figura 25. O uso do DateHelper para manipulação das datas.*




Perceba que com o uso do helper continuamos gerando o objeto Consulta e também exibimos a data no formato correto. A diferença agora é que esse código está mais organizado e com as responsabilidades definidas. Mas, sempre podemos melhorar.

## Métodos estáticos

Geralmente, não existe nenhuma propriedade ou construtor para uma classe Helper que justifique a criação de uma instância para uso dos métodos. Vale ressaltar que toda classe no ES6 tem por padrão o construtor vazio.

```
constructor() {}
```

Criar uma instância de uma classe Helper toda vez que for utiliza-la causa um impacto muito grande na memória, em ambientes escalados pode causar a lentidão e até derrubar o servidor. Para solucionar esse problema, utilizamos um recurso chamado método estático. Ele permite que os métodos da classe sejam acessados diretamente, sem a necessidade de uma instância do objeto para uso. A palavra reservada static, antes do nome do método. Veja a classe DateHelper utilizando métodos estáticos:

```
class DateHelper {
    static textoParaData(texto) {
        return new Date(...
            texto
                .split('-')
                .map((item, indice) => item - (indice % 2)));
    }

    static dataParaTexto(data) {
```

```
        return data.getDate()
            + "/" + (data.getMonth() + 1)
            + "/" + data.getFullYear();
    }
}
```

O próximo passo é adequar a classe controller para chamada do método estático:

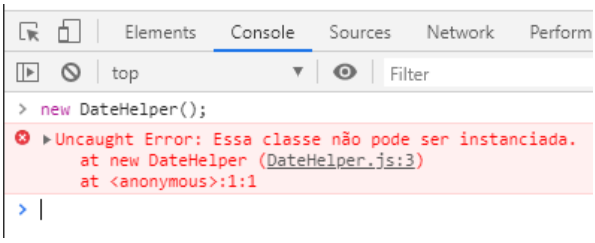
```
let consulta = new Consulta(
    this._inputNome.value,
    DateHelper.textoParaData(this._inputData.value),
    this._inputPeso.value,
    this._inputAltura.value
);
```

Mas, ainda existe um pequeno problema em nosso código. A classe DateHelper possui o construtor padrão, mas ela não pode ser gerada uma instância. Uma maneira de resolver isso é lançando uma exceção caso alguém invoque uma instância do nosso helper. Confira o resultado:

```
class DateHelper {
    constructor() {
        throw new Error("Essa classe não pode ser instanciada.");
    }

    static textoParaData(texto) {
        return new Date(...
            texto
                .split('-')
                .map((item, indice) => item - (indice % 2)));
    }

    static dataParaTexto(data) {
        return data.getDate()
            + "/" + (data.getMonth() + 1)
            + "/" + data.getFullYear();
    }
}
```

Vamos conferir o resultado caso uma instância seja invocada, na Figura 26.

*Figura 26. Erro ao tentar criar uma instância da classe DateHelper.*



## Template Strings

Existe sempre uma maneira de melhorarmos nosso código. Desenvolver um algoritmo não é diferente de escrever uma redação, nas suas devidas proporções. Sempre podemos deixar o código menos verboso e mais claro. A template string auxilia nessa tarefa. Ela possibilita o uso de uma expression language para facilitar o trabalho de exibir valores em uma string formatada.

Vamos pegar o método dataParaTexto da classe DateHelper:

```
static dataParaTexto(data) {
    return data.getDate()
        + "/" + (data.getMonth() + 1)
        + "/" + data.getFullYear();
}
```

Alguns cuidados são necessários para compor a string de exibição, além da concatenação das strings com as propriedades. A soma entre o mês e 1 são colocadas entre parênteses para evitar quaisquer problemas nas operações. Com o uso da template string isso é facilitado, e esses problemas não ocorrem. Veja como fica o método com a template string:

```
static dataParaTexto(data) {
    return `${data.getDate()}/${data.getMonth() + 1}/
            ${data.getFullYear()}`;
}
```

Primeiro ponto é que um template string utiliza a crase (backtick) para determinar o conteúdo da string. Os valores de uma propriedade podem ser exibidos na string através da interpolação do conteúdo apresentado dentro do ${}. Agora exibir informações através da string é muito mais prático. Lembrando que dentro do interpolador da expression language, pode ser inserido qualquer operação existente no JavaScript.

## Validação do formato da data

Alguns cuidados ainda precisam ser tomados no código. O input do tipo date, no HTML5, prove uma data no formato yyyy-mm-dd. Porém, em nenhum momento estamos validando essa entrada no método de conversão de textoParaData. Vamos utilizar o objeto de expressão regular do JavaScript para fazer isso. Declarar uma expressão regular é fácil, basta colocar a expressão desejada entre barras (/ /). Vamos fazer a validação da entrada:

```
if(!/^\d{4}-\d{2}-\d{2}$/.test(texto))
    throw new Error('O formato correto é yyyy-mm-dd');
```

O if deve ser declarado antes do retorno do método. Informamos que o texto recebido de possuir 4 dígitos seguindo por um traço, mais 2 dígitos, outro traço e mais 2 dígitos. O acento circunflexo e o sinal de cifrão, informam que nenhum caractere a mais será aceito. Agora nossa classe DateHelper está completa:

```
class DateHelper {
    constructor() {
        throw new Error("Essa classe não pode ser instanciada.");
    }

    static textoParaData(texto) {
        if(!/^\d{4}-\d{2}-\d{2}$/.test(texto))
            throw new Error('O formato correto é yyyy-mm-dd');
        return new Date(...
            texto
                .split('-')
                .map((item, indice) => item - (indice % 2)));
    }

    static dataParaTexto(data) {
        return `${data.getDate()}/${data.getMonth() + 1}/
                ${data.getFullYear()}`;
    }
}
```

## O que aprendemos

- Conceito de helpers;
- Métodos estáticos;
- Template Strings;
- Validação com Expressão Regular.

O código atualizado encontra-se em https://github.com/amasiero/app-nutricao-js/tree/v6

# Construindo a lista de consultas

Nosso sistema está preparado para trabalhar com uma consulta. Entretanto, nossa nutricionista precisa que o suporte seja dado para diversas consultas. Podemos pensar em utilizar um simples array, mas isso limita as operações que precisamos sobre as consultas armazenadas.

Uma boa prática nesse caso é criar um modelo de lista para consultas, o que chamamos de ListModel. Vamos criar uma classe chamada ListaConsultas.js na subpasta models, que terá uma lista iniciada com zero elementos:

```
class ListaConsultas {
    constructor() {
        this._consultas = [];
    }

    adiciona(consulta) {
        this._consultas.push(consulta);
    }
}
```

Perceba o encapsulamento da lista, isso significa que ela só pode ser acessada através dos métodos da classe. Dessa forma, vamos criar a propriedade get para acessar a lista:

```
class ListaConsultas {
    constructor() {
        this._consultas = [];
    }

    adiciona(consulta) {
        this._consultas.push(consulta);
    }

    get consultas() {
        return this._consultas;
    }
}
```

O próximo passo é incluir a lista no index.html para utilizarmos nas demais classes:

```
<script type="text/javascript"
src="js/app/models/Consulta.js"></script>
<script type="text/javascript"
src="js/app/models/ListaConsultas.js"></script>
<script type="text/javascript"
src="js/app/helpers/DateHelper.js"></script>
```

```
<script type="text/javascript"
src="js/app/controllers/ConsultaController.js"></script>
<script type="text/javascript">
  let consultaController = new ConsultaController();
</script>
```

Vamos atualizar na sequência a classe ConsultaController para ter uma propriedade de ListaConsultas, que armazenará todas as consultas exibidas na tabela:

```
class ConsultaController {
    constructor() {
        let $ = document.querySelector.bind(document);
        this._inputNome = $('#nome');
        this._inputData = $('#data');
        this._inputPeso = $('#peso');
        this._inputAltura = $('#altura');
        this._listaConsultas = new ListaConsultas();
    }

    adiciona(evento) {
        evento.preventDefault();
        let consulta = new Consulta(
            this._inputNome.value,
            DateHelper.textoParaData(this._inputData.value),
            this._inputPeso.value,
            this._inputAltura.value
        );
        this._listaConsultas.adiciona(consulta);
    }
}
```

Algumas melhorias podem ser feitas no código. Perceba que o método adiciona está ganhando um grande volume de responsabilidade que pode ser compartilhada. E ainda, após adicionar a consulta na lista, não apagamos o formulário. Vamos criar dois métodos com underline na frente do nome, significando que eles são utilizados apenas dentro da classe. Simulando métodos privados:

```
class ConsultaController {

    //Código omitido ...

    _criaConsulta() {
        return new Consulta(
            this._inputNome.value,
            DateHelper.textoParaData(this._inputData.value),
            this._inputPeso.value,
            this._inputAltura.value
        );
    }
```

```
    _limpaFormulario() {
        this._inputNome.value = "";
        this._inputData.value = "";
        this._inputPeso.value = "";
        this._inputAltura.value = "";

        this._inputNome.focus();
    }
}
```

Agora os métodos da classe estão com as devidas responsabilidades e algumas foram apenas por uma questão de organização, afinal é interna a classe. Mas, precisamos aplicar a programação defensiva para blindarmos a lista de consultas. Na classe ListaConsultas, altere a propriedade get para:

```
get consultas() {
    return [].concat(this._consultas);
}
```

Pronto, nossa classe agora está com as devidas precauções tomadas garantindo a integridade da informação.

## O que aprendemos

- ListModel Design Pattern
- Métodos privados simulados
- Blindagem da lista

O código atualizado encontra-se em https://github.com/amasiero/app-nutricao-js/tree/v7

# E o usuário, view?

Nós já temos a nossa camada model e controller do modelo MVC. Entretanto, o precisamos fazer a parte da view. Você pode estar se perguntando sobre o HTML, e você tem razão. Ele é parte da camada de view, mas podemos deixar a interação ainda melhor trazendo para uma classe algumas partes interativas do HTML. Alguns frameworks, como o React, também adotam esse tipo de estratégia. Na subpasta views, vamos criar a classe ConsultasView, que será responsável por criar a tabela:

```
<table class="table table-bordered">
  <thead class="thead-dark">
    <tr>
      <th scope="col">Nome</th>
      <th scope="col">Data da Consulta</th>
      <th scope="col">Peso (kg)</th>
      <th scope="col">Altura (m)</th>
      <th scope="col">IMC</th>
    </tr>
  </thead>
  <tbody>

  </tbody>
</table>
```

Retiramos esse trecho de código do HTML. O resultado deve ser o mesmo apresentado na Figura 27.

*Figura 27. Resultado da página index.html após remoção da tabela.*


A classe ConsultasView terá um método chamado template, que retornará o código da tabela. O mesmo removido anteriormente.

```
class ConsultasView {
    template() {
        return `
        <table class="table table-bordered">
        <thead class="thead-dark">
            <tr>
                <th scope="col">Nome</th>
                <th scope="col">Data da Consulta</th>
                <th scope="col">Peso (kg)</th>
                <th scope="col">Altura (m)</th>
                <th scope="col">IMC</th>
            </tr>
        </thead>
        <tbody>

        </tbody>
        </table>
        `;
    }
}
```

Agora, é necessário fazer o vínculo entre a classe view e a controller. Lembre-se a model nunca conversa direto com a view, tudo deve passar pelo controller. Vamos criar uma propriedade da view na classe controller:

```
class ConsultaController {
    constructor() {
        let $ = document.querySelector.bind(document);
        this._inputNome = $('#nome');
        this._inputData = $('#data');
        this._inputPeso = $('#peso');
        this._inputAltura = $('#altura');
        this._listaConsultas = new ListaConsultas();
        this._consultasView = new ConsultasView();
    }
// Código omitido...
```

No local onde a tabela deve ficar, vamos criar uma div para vincular com o código da tabela gerado pelo template.

```
<div id="consultasView"></div>
```

A associação entre a div e a classe será feita através do construtor que receberá o DOM que deve ser renderizado.

```
class ConsultasView {
    constructor(elemento) {
        this._elemento = elemento;
    }
// Código omitido...
```

Agora devemos passar o elemento para a classe da view através do controller. Na sequência, vamos chamar o método update para atualizar a nossa view. Veja a atualização da classe controller:

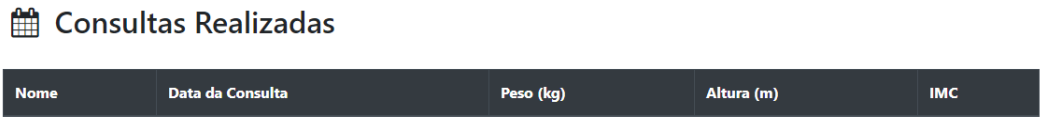
```
class ConsultaController {
    constructor() {
        let $ = document.querySelector.bind(document);
        this._inputNome = $('#nome');
        this._inputData = $('#data');
        this._inputPeso = $('#peso');
        this._inputAltura = $('#altura');
        this._listaConsultas = new ListaConsultas();
        this._consultasView = new
ConsultasView($('#consultasView'));
        this._consultasView.update();
    }
// Código omitido...
```

Já que utilizaremos o método update para atualizar a nossa view, o método template pode ser privado a partir desse momento.

```
update() {
    this._elemento.innerHTML = this._template();
}

_template() {
    return `
        <table class="table table-bordered">
        <thead class="thead-dark">
            <tr>
                <th scope="col">Nome</th>
                <th scope="col">Data da Consulta</th>
                <th scope="col">Peso (kg)</th>
                <th scope="col">Altura (m)</th>
                <th scope="col">IMC</th>
            </tr>
        </thead>
        <tbody>

        </tbody>
        </table>
    `;
}
```

A Figura 28 apresenta a renderização da view no navegador.

*Figura 28. Tabela renderizada pela classe view.*



A propriedade innerHTML possibilita a inclusão de quaisquer tags HTML como se fosse uma string. Assim, é mais fácil para criar o código todo.

## Deixando o template dinâmico

Para deixar o template dinâmico é necessário passar para o método update o modelo que contém todas as informações que serão renderizadas no através do template. Veja como fica a chamada do método update:

```
this._consultasView.update(this._listaConsultas);
```

Agora vamos atualizar a classe ConsultasView, e através do uso do map, construir as linhas da tabela de acordo com a ListaConsultas. Veja o código:

```
class ConsultasView {
    constructor(elemento) {
        this._elemento = elemento;
    }

    update(model) {
        this._elemento.innerHTML = this._template(model);
    }

    _template(model) {
        return `
            <table class="table table-bordered">
            <thead class="thead-dark">
                <tr>
                    <th scope="col">Nome</th>
                    <th scope="col">Data da Consulta</th>
                    <th scope="col">Peso (kg)</th>
                    <th scope="col">Altura (m)</th>
                    <th scope="col">IMC</th>
                </tr>
            </thead>
            <tbody>
                ${model.consultas.map(c => `
                    <tr>
                        <td>${c.nome}</td>
                        <td>${DateHelper.dataParaTexto(c.data)}</td>
                        <td>${c.peso}</td>
                        <td>${c.altura}</td>
                        <td>${c.imc}</td>
                    </tr>
                `)}
            </tbody>
            </table>
        `;
    }
}
```

Quando o _template for retornar a string, terá que processar o trecho do return primeiramente, e depois retornar a template string. Para cada consulta será criada uma lista, cada uma com as tags <tr> e os dados cadastrados. Estamos varrendo a lista e para um objeto Consulta, estamos criando um array, mas o novo elemento será uma string com os dados. No entanto, por enquanto, o retorno será um array. Por isso, adicionaremos o join(). Assim, faremos a junção com uma string em branco, convertendo o array em uma string aceita na propriedade innerHTML:

```
${model.consultas.map(c => `
    <tr>
        <td>${c.nome}</td>
        <td>${DateHelper.dataParaTexto(c.data)}</td>
        <td>${c.peso}</td>
        <td>${c.altura}</td>
        <td>${c.imc.toFixed(2)}</td>
    </tr>
`).join('')}
```

Por fim, devemos atualizar nossa view também quando for incluída uma nova consulta:

```
adiciona(evento) {
    evento.preventDefault();
    this._listaConsultas.adiciona(this._criaConsulta());
    this._consultasView.update(this._listaConsultas);
    this._limpaFormulario();
}
```

Na Figura 29 é possível conferir o resultado após a inclusão de um paciente no sistema de consulta.

*Figura 29. Inclusão realizada com o auxílio do template dinâmico.*

## O que aprendemos

- Classe view
- Conexão view controller
- Propriedade innerHTML
- Template dinâmico

O código atualizado encontra-se em https://github.com/amasiero/app-nutricao-js/tree/v8

# Melhorando a experiência do usuário

A aplicação está funcionando direitinho, conforme o esperado até o momento. Mas, a experiência do usuário pode ser melhorada quando uma nova consulta for cadastrada, informar ao usuário. Essa é uma das heurísticas de Nielsen para garantir uma melhor usabilidade do sistema. Vamos começar criando um modelo para mensagem:

```
class Mensagem {
    constructor(texto = '') {
        this._texto = texto;
    }

    get texto() {
        return this._texto;
    }

    set texto(texto) {
        this._texto = texto;
    }
}
```

Nessa classe criamos as propriedades set e get para manipular o atributo _texto. Um fator interessante é o uso de uma atribuição no parâmetro do método constructor. Essa técnica determina um valor padrão para o parâmetro, tornando-o opcional durante a declaração.

O próximo passo é criar a classe MensagemView que receberá as mensagens para exibição na tela sempre que preciso. Mas antes, vamos criar um modelo mensagem no controller:

```
class ConsultaController {
    constructor() {
        let $ = document.querySelector.bind(document);
        this._inputNome = $('#nome');
        this._inputData = $('#data');
        this._inputPeso = $('#peso');
        this._inputAltura = $('#altura');
        this._listaConsultas = new ListaConsultas();

        this._consultasView = new
ConsultasView($('#consultasView'));
        this._consultasView.update(this._listaConsultas);
        this._mensagem = new Mensagem();
    }
// Código omitido...
```

E também já instanciar a mensagem após adicionar uma nova consulta:

```
adiciona(evento) {
    evento.preventDefault();
    this._listaConsultas.adiciona(this._criaConsulta());
    this._mensagem.texto = 'Consulta adicionada com sucesso.'
    this._consultasView.update(this._listaConsultas);
    this._limpaFormulario();
}
```

Agora, criaremos a classe MensagemView, confira:

```
class MensagemView {
    constructor(elemento) {
        this._elemento = elemento;
    }

    update(model) {
        this._elemento.innerHTML = this._template(model);
    }

    _template(model) {
        return `<p class="alert alert-info">${model.texto}</p>`;
    }
}
```

Antes de vincularmos a view com o controller, devemos criar o espaço no HTML para inserir a mensagem. Acima da tag form insira:

```
<div id="mensagemView" class="col-12"></div>
```

Nesse momento, é possível trabalhar com o controller para apresentar a mensagem de sucesso da operação. Começando pelo construtor:

```
constructor() {
    // Código omitido ...
    this._mensagem = new Mensagem();
    this._mensagemView = new MensagemView($('#mensagemView'));
}
```

E depois a invocação do método com a mensagem após a inclusão da consulta.

```
adiciona(evento) {
    evento.preventDefault();

    this._listaConsultas.adiciona(this._criaConsulta());
    this._mensagem.texto = 'Consulta adicionada com sucesso.'
    this._mensagemView.update(this._mensagem);

    this._consultasView.update(this._listaConsultas);
    this._limpaFormulario();
}
```

Confira o resultado, através da Figura 30, após incluir uma consulta no sistema.

*Figura 30. Mensagem de sucesso aparecendo após a inclusão da consulta no sistema.*


## Reutilizando código através da Herança de classes

Você percebeu que as duas classes da camada view possuem uma estrutura bem parecida. Sendo que os métodos constructor e update executam o mesmo código. Já deve estar pensando, vamos enxugar esse código. Reaproveitar sempre, repetir jamais, esse é o nosso lema. Em orientação à objetos temos uma técnica chamada Herança, para aproveitarmos o código nesse tipo de cenário. Sendo assim, vamos criar uma classe View, mais abstrata que as ConsultasView e MensagemView.

```
class View {
    constructor(elemento) {
        this._elemento = elemento;
    }

    update(model) {
        this._elemento.innerHTML = this._template(model);
    }
}
```

Agora, vamos fazer com que MensagemView seja uma filha da classe View. Em outras palavras herde os métodos e propriedades de View.

```
class MensagemView extends View {
    _template(model) {
        return `<p class="alert alert-info">${model.texto}</p>`;
    }
}
```

Perceba que só foi necessário incluir a palavra extends após o nome da classe e em seguida o nome da classe mãe, no exemplo View. A mesma coisa deve ser feita para ConsultasView. E não se esqueça de adicionar o script no arquivo index.html:

```
<script type="text/javascript"
src="js/app/models/Consulta.js"></script>
<script type="text/javascript"
src="js/app/models/ListaConsultas.js"></script>
<script type="text/javascript"
src="js/app/models/Mensagem.js"></script>
<script type="text/javascript"
src="js/app/helpers/DateHelper.js"></script>
<script type="text/javascript" src="js/app/views/View.js"></script>
<script type="text/javascript"
src="js/app/views/ConsultasView.js"></script>
<script type="text/javascript"
src="js/app/views/MensagemView.js"></script>
<script type="text/javascript"
src="js/app/controllers/ConsultaController.js"></script>
<script type="text/javascript">
  let consultaController = new ConsultaController();
</script>
```

Apesar de tudo estar funcionando de maneira adequada, já que a classe filha enxerga os métodos da classe mãe, é uma boa prática declarar o construtor nas classes filhas. Dentro deles efetuar uma chamada para o método super. O método super invoca o construtor da classe mãe. Esse super é de super class, a classe superior na hierarquia, a classe mãe. O código deve ficar assim:

```
class MensagemView extends View {
    constructor(elemento) {
        super(elemento);
    }
    _template(model) {
        return `<p class="alert alert-info">${model.texto}</p>`;
    }
}
```

Para ambas classes esse trabalho deve ser realizado. Outra boa prática que deve ser adota é a declarado do método template na classe View. Mas não iremos repetir o código assim? Não, cada classe tem uma implementação diferente. No caso de View declarar o template é devido a invocação do método dentro de update. Só que o template de View deve lançar um Error informando para implementar esse método na classe filha. Confira a implementação desse método:

```
_template(model) {
    throw new Error('O método template deve ser implementado.')
}
```

Pronto, as classes agora estão construídas de maneira adequada para uma modelagem de sistemas.

## O que aprendemos

- Classe Mensagem
- Parâmetro opcional
- Propriedade set
- Classe MensagemView
- Herança
- Super class

O código atualizado encontra-se em https://github.com/amasiero/app-nutricao-js/tree/v9