Lecom BPM – 5.50

Documentação de uso da API JavaScript com formulário

Introdução

Este documento tem por objetivo apresentar alguns padrões e funções JavaScript para uso nos formulários do Lecom BPM versão 5.30.

Essas funções são alternativas para manipulação de elementos e segue um novo padrão da API de campo.

1. Acessando a API do Form no script:

Para instanciar e utilizar a variável da API do Form de processo em seu script, basta executar o seguinte comando:

```
// Antes era utilizado da seguinte forma:
var formAPI = Lecom.api.ProcessAPI.currentProcess().form();

// Atualmente é:
Form;
```

É com este comando que chamaremos todas as funções da API.

2. Componentes que podem ser manipulados pela API do Form:

Este componente encapsula o escopo global do Form, ele é acessado diretamente pela variável **formAPI** e é o ponto de partida para acessar todos os outros componentes da API.

Propriedades:

fields – Lista as APIs de campo para cada campo básico do Form (que não seja parte de um grid). O maior detalhamento sobre o uso desta propriedade é descrito abaixo.
groups – Lista as APIs de grupo para cada grupo do Form. O maior detalhamento sobre o uso desta propriedade é descrito abaixo.
actions – Lista as APIs de ação para as ações relacionadas diretamente ao Form (que não estejam relacionadas a um campo). O maior detalhamento sobre o uso desta propriedade é descrito abaixo.
errors – Representa todos os erros de validação presente nos campos do Form. É definido como um objeto, onde cada chave representa o ID de um campo e os valores são um array de mensagens de erro que serão exibidas na tela, embaixo do campo em vermelho.

Exemplo:

```
// Definindo um erro no campo com ID NOME:
Form.errors({ NOME: 'Campo inválido' }).apply();
```

readOnly: valor *booleano* que define em tela o Form em estado “somente leitura”.

Exemplo:

```
// Definindo o Form como “somente leitura”:
Form.readOnly(true).apply();
```

waitingForAction: valor booleano que define em tela se o Form está esperando uma resposta do servidor. O Form é apresentado então no estado “carregando”.

Exemplo:

```
// Definir o form como “carregando”.
Form.waitingForAction(true).apply();
```

3. Grupos

Componente de grupo básico de um Form, é acessado através função **groups** do componente **Form** da API. Possui propriedades e funções específicas para a manipulação de todos os tipos de campo e grid.

Exemplo de acesso a um componente deste tipo:

```
Form.groups('ID_GROUP');
```

3.1 Propriedades de grupos

fields – Retorna todos os campos de um grupo específico;

Exemplo:

```
// Retorna todos os campos que pertencem ao grupo informado
Form.groups('ID_GROUP').fields();

// Retorna um campo específico
Form.groups('ID_GROUP').fields('ID_FIELD');
```

Obs.: Ao informar o ID de um campo que pertença a outro grupo nenhuma ação será realizada e a informação de ‘undefined’ será exibida no console do browser.

```
Form.groups('ID_GROUP').fields('ID_FIELD_DE_OUTRO_GRUPO');
```

grids – Retornar todas as grids de um grupo específico;

Exemplo:

```
// Retorna todas as grids que pertencem ao grupo informado
Form.groups('ID_GROUP').grids();

// Retorna apenas a grid informada do grupo específico
Form.groups('ID_GROUP').grids('ID_GRID');
```

expanded – Método com valor booleano que permite expandir e/ou fechar um grupo no formulário.

Exemplo:

```
// Retorna true e/ou false
Form.groups('ID_GROUP').expanded();

// Expandir o grupo
Form.groups('ID_GROUP').expanded(true);

// Fechar o grupo
Form.groups('ID_GROUP').expanded(false);
```

Obs.: Ao tentar fechar um agrupador que não esteja com a opção “Permitir expandir e contrair grupo” configurada no StudioWeb (Campos do Modelo), nenhuma ação deverá ser realizada e uma mensagem será apresentada no console do browser.

disabled – Método com valor booleano que permite tornar um grupo e seus campos bloqueados e/ou desbloqueados no formulário.

Exemplo:

```
// Retorna true e/ou false
Form.groups('ID_GROUP').disabled();

// Torna o grupo e seus campos e grids bloqueados
Form.groups('ID_GROUP').disabled(true).apply();

// Torna o grupo e seus campos e grids desbloqueados
Form.groups('ID_GROUP').disabled(false).apply();
```

visible – Método com valor booleano que permite tornar um grupo e seus campos invisíveis e/ou visíveis.

Exemplo:

```
// Retorna true e/ou false
Form.groups('ID_GROUP').visible();
```

```
// Torna o grupo e seus campos e grids visíveis
Form.groups('ID_GROUP').visible(true).apply();

// Torna o grupo e seus campos e grids invisíveis
Form.groups('ID_GROUP').visible(false).apply();
```

4. Campo

Componente de campo básico de um Form, é acessado através função **fields** do componente **Form** da API. Possui propriedades e funções específicas para os tipos de campo **Grid** e **Autocomplete** (Lista).

Exemplo de acesso a um componente deste tipo:

```
// API para o campo com ID NOME.
var nomeAPI = Form.fields('NOME');
```

5. Campo de Grid

Componente de campo que pertence a um Grid, é acessado através da função **fields** de um componente de campo do tipo **Grid** da API.

Exemplo de acesso a um componente deste tipo:

```
// API para o campo com ID PRECO do grid com ID GRID1.
var precoGridAPI = Form.grids('GRID1').fields('PRECO');
```

6. Propriedades de campos:

actions – Lista as APIs de ação para as ações relacionadas ao Campo. O maior detalhamento sobre o uso desta propriedade é descrito abaixo.
disabled – Valor do tipo booleano que determina se o campo está bloqueado ou não.

Exemplo:

```
// Definindo o campo com ID NOME como bloqueado.
Form.fields('NOME').disabled(true).apply();
```

errors – Array de mensagens de erro de validação que são exibidas em vermelho embaixo do campo. Também aceita uma mensagem de erro como string.

Exemplo:

```
// Apresentando erros de presença e validação no campo com ID CPF.
Form.fields('CPF').errors(['Valor obrigatório', 'CPF Inválido']).apply();
```

helpText – Valor string que determina o texto de ajuda para o campo.

Exemplo:

```
// Definindo um texto de ajuda para o campo com ID NOME.
Form.fields('NOME').helpText('Nome completo do usuário').apply();
```

label – Valor string que representa a label do campo.

Exemplo:

```
// Definindo uma nova label para o campo com ID NOME:
Form.fields('NOME').label('Nome completo').apply();
```

mask – Valor string para definir uma máscara no campo. Ela pode ser definida com valor de máscara pré-definida, como “cpf” ou “cnpj”, ou uma máscara customizada. Apenas os campos dos tipos LINHA_DE_TEXTO, aceitam esta propriedade. A lista de máscaras pré-definidas, e as instruções de como montar uma máscara customizada estão disponíveis na seção Dicas.

Exemplo:

```
// Definindo máscara de CPF para o campo com ID CPF:
Form.fields('CPF').mask('cpf').apply();
```

```
// Definindo máscara customizada para o campo com ID CEP:
Form.fields('CEP').mask('99.999-999').apply();
```

maxLength – Valor do tipo number usado para definir o comprimento máximo de um campo.

Exemplo:

```
// Definindo maxLength de 2 caracteres para o campo com ID IDADE:
Form.fields('IDADE').maxLength(2).apply();
```

readOnly – Valor do tipo booleano que determina se o campo é somente leitura ou não.

Exemplo:

```
// Definindo o campo com ID NOME como somente leitura:
Form.fields('NOME').readOnly(true).apply();
```

linebreak – Valor do tipo string que determina o tipo de quebra de linha do campo.

Opções de quebra de linha

1. NENHUM
2. SIMPLES
3. DUPLA
4. BARRA_SEPARADORA

Exemplo:

```
// Incluindo quebra de linha "BARRA SEPARADORA"  para o campo com ID NOME:
Form.fields('NOME').linebreak('BARRA_SEPARADORA').apply();
```

value – Esta propriedade representa o valor do campo e cada tipo de campo deve receber um formato de dado, exemplo:

1. **Linha de texto:** String

```
// Definindo um novo valor para o campo:
Form.fields('LINHA_DE_TEXTO').value('texto').apply();
```

2. **Caixa de texto:** String

```
// Definindo um novo valor para o campo:
Form.fields('CAIXA_DE_TEXTO').value('texto').apply();
```

3. **Inteiro:** Int

```
// Definindo um novo valor para o campo:
Form.fields('INTEIRO').value(150).apply();
```

4. **Número decimal:** Number

```
// Definindo um novo valor para o campo:
Form.fields('DECIMAL').value(150.50).apply();
```

5. **Monetário:** Number

```
// Definindo um novo valor para o campo:
Form.fields('MONETARIO').value(1500.50).apply();
```

6. **Data:** String

```
// Definindo um novo valor para o campo:
Form.fields('DATA').value('06/03/2018').apply();
```

7. **Lista:** String

```
// Definindo um novo valor para o campo:
Form.fields('LISTA').value('Opcao 1').apply();
```

8. **Radio button:** String

```
// Definindo um novo valor para o campo:
Form.fields('RADIO').value('Opcao 1').apply();
```

9. **Checkbox:** String

```
// Definindo um novo valor para o campo:
Form.fields('CHECKBOX').value('Checado').apply();
```

visible – Valor do tipo booleano que determina se o campo é visível ou não.

Exemplo:

```
// Tornando o campo com ID NOME invisível:
Form.fields('NOME').visible(false).apply();
```

7. Propriedades de campo do tipo Grid:

fields – Lista as APIs de campo para cada campo associado ao Grid.
columns – Valor do tipo array com a lista das definições das colunas do Grid. Cada item desta lista é um objeto com 3 propriedades:

1. **label** – Título da coluna.
2. **name** – Identificador da coluna, normalmente representado pelo ID do campo do grid relacionado ao valor desta coluna.
3. **value** – Função cujo retorno define o valor da coluna em cada linha do grid. Recebe o dataRow como parâmetro. Esta propriedade é opcional, e o valor entrado pelo campo do grid com o ID relacionado à propriedade name.

8. Propriedades de campos do tipo Autocomplete:

options – Valor do tipo array que representa as opções da lista do campo. Cada item deste array é um objeto com chaves name, para o valor que é exibido no campo e value, que é o valor que será enviado para o serbidor nas ações de submissão do Form (Salvar, Segue Etapa ou Finalizar Etapa).

Exemplo:

```
// Retornando as opções do campo de ID LISTA em uma variável.
var opcoesLista = Form.fields('LISTA').options();
```

9. Funções de campos:

removeMask – Remove a máscara de um campo do tipo linha de texto que tem uma máscara definida.

Exemplo:

```
var nomeAPI = Form.fields('NOME');

// Remover máscara.
nomeAPI.removeMask().apply();
```

focus – Ao ser chamada, o campo recebe o foco do cursor, para que o usuário possa digitar no campo sem a necessidade de selecioná-lo com o mouse.

Exemplo:

```
var nomeAPI = Form.fields('NOME');

// Colocar foco no campo com ID NOME após torná-lo visível.
nomeAPI.visible(true).apply().then(() => { nomeAPI.focus(); });

// Browser IE (Internet Explorer):
nomeAPI.visible(true).apply().then( function() {nomeApi.focus()});
```

10. Funções de campos do tipo Grid:

visible – Get/set visibilidade da grid

Exemplo:

```
// Get visible
var gridAPI = Form.grids('GRID');
```

```
gridAPI.visible(); // retorna true/false

// Set visible
var gridAPI = Form.grids ('GRID');
gridAPI.visible(true/false).apply();
```

visible dos campos da grid – Get/set visibilidade de campos da grid

Exemplo:

```
// Get visible
var gridAPI = Form.grids('GRID');
gridAPI.fields('CAMPO').visible(); // retorna true/false

// Set visible
var gridAPI = Form.grids ('GRID');
gridAPI.fields.('CAMPO').visible(true/false).apply();
```

visible das colunas da grid – Get/set visibilidade de colunas da grid

Exemplo:

```
// Get visible
var gridAPI = Form.grids ('GRID');
gridAPI.columns('COLUNA').visible(); // retorna true/false

// Set visible
var gridAPI = Form.grids ('GRID');
gridAPI.columns('COLUNA').visible(true/false).apply();
```

dataRows – Retorna os dataRows do campo Grid em um array. Recebe um parâmetro de filtro que pode ter duas formas: posição do dataRow no Grid (começando por 0) ou uma função que recebe cada dataRow como parâmetro e que, se ela retorna true para um determinado dataRow, ele é retornado.

Exemplo:

```
var gridAPI = Form.grids ('GRID');

// Retorna todos os dataRows do campo grid com ID GRID.
var dataRows = gridAPI.dataRows();

// Retorna o dataRow do campo grid na segunda posição (1):
var secondDataRow = gridAPI.dataRows(1);

// Retorna os dataRows do campo grid cujo campo PRECO possui o valor maior que $20,00:
```

```
var expensiveDataRows = gridAPI.dataRows((dataRow) => {
        return dataRow.PRECO > 20.0;
});
```

insertDataRow – insere um novo dataRow ao campo Grid. Recebe como parâmetro o objeto de dataRow a ser inserido.

Exemplo:

```
var gridAPI = Form.grids('GRID');

// Insere um novo dataRow para o GRID que possui campos com IDs NOME, PRECO_UNITARIO e QUANTIDADE.
gridAPI.insertDataRow({ NOME: 'Item 1', PRECO_UNITARIO: 20.99, QUANTIDADE: 10 });
```

updateDataRow – Atualiza os valores de um dataRow do campo Grid e recebe como parâmetro o objeto de dataRow a ser atualizado. A referência de qual dataRow existe será atualizado por esta função que é determinada pelo valor da propriedade **id** de seu objeto, que pode ser obtido utilizando a função **dataRows**, como mostrado no exemplo abaixo.

Exemplo:

```
var gridAPI = Form.grids ('GRID');

// Recupera o id do dataRow do item de nome 'Item 1'.
var dataRowId = gridAPI.dataRows((dataRow) => {
        return dataRow.NOME === 'Item 1';
});

// Altera o preço unitário deste dataRow para $30,99.
gridAPI.updateDataRow({
        id: dataRowId, PRECO_UNITARIO:   30.99
});

// IE – Internet Explorer (Recupera o ID do dataRow do item de nome ‘Item 1’).
var dataRowId = gridAPI.dataRows(function(dataRow){
        return dataRow.NOME === 'Item 1';
});
```

removeDataRow – Remove um dataRow do campo Grid. Recebe como parâmetro (identificador do dataRow) que deve ser removido. Este indicador pode ser obtido utilizando a unção dataRows, como o exemplo abaixo:

Exemplo:

```
var gridAPI = Form.grids ('GRID');



// Remove todos os dataRows cujo preço é maior que $20,00.
var dataRowsToRemove = gridAPI.dataRows((dataRow) => {
	return dataRow.PRECO_UNITARIO > 20;
});

// Usa o iterador forEach para chamar o removeDataRow para cada id da lista em dataRowsToRemove
dataRowsToRemove.forEach((dataRow) => {
	gridAPI.removeDataRow(dataRow.id);
});

// Exemplo no (IE) – Internet Explorer
var dataRowsToRemove = gridAPI.dataRows(function(dataRow){
	return dataRow.PRECO_UNITARIO > 20;
});

dataRowsToRemove.forEach(function(dataRow){
	gridAPI.removeDataRow(dataRow.id);
});
```

addColumn – Adiciona uma nova coluna ao grid.

Exemplo:

```
var gridAPI = Form.grids ('GRID');

// Adiciona a coluna PRECO_TOTAL
gridAPI.addColumn({
	name: 'PRECO_TOTAL',
	label: 'Preço total'
}).apply();
```

Se passado **true** como segundo parâmetro, a coluna será adicionada na primeira coluna.

```
var gridAPI = Form.grids ('GRID');

// Adiciona a coluna PRECO_TOTAL
gridAPI.addColumn({
	name: 'PRECO_TOTAL',
	label: 'Preço total'
```

```
}, true).apply();
```

editColumn – Edita uma coluna existente do grid. Recebe como parâmetro o objeto de coluna que vai ser atualizado. A referência de qual coluna existente vai ser alterada por essa função é determinada pelo valor da propriedade **name** de seu objeto. Pode ser utilizada para alterar o label da coluna e a função que produz o seu valor de exibição.

Exemplo:

```
var gridAPI = Form.grids ('GRID');

// Altera o label da coluna PRECO_UNITARIO do grid com ID GRID.
gridAPI.editColumn({
        name: 'PRECO_UNITARIO',
        label: 'Preço unitário'
}).apply();
```

removeColumn – Remove uma coluna do grid. Recebe como parâmetro a propriedade **name**, identificadora da coluna que deve ser removida.

Exemplo:

```
var gridAPI = Form.grids ('GRID');

// Remove a coluna VALOR_OCULTO do grid
gridAPI.removeColumn('VALOR_OCULTO').apply();
```

sum – É um método totalizador, que possibilita obter o total de uma coluna dos campos Monetário, Numérico e Inteiro.

Exemplo:

```
var gridAPI = Form.grids('GRID');

gridAPI.columns("NUMBER_GRID").sum();
```

Min – É um método que possibilita obter o valor mínimo de uma coluna para os campos Monetário, Numérico e Inteiro.

Exemplo:

```
var gridAPI = Form.grids('GRID');
```

```
gridAPI.columns(“NUMBER_GRID”).min();
```

Max – É um método que possibilita obter o valor máximo de uma coluna para os campos Monetário, Numérico e Inteiro.

Exemplo:

```
var gridAPI = Form.grids('GRID');

gridAPI.columns(“NUMBER_GRID”).max();
```

11. Função format para coluna da grid

A formatação da coluna foi implementada da seguinte forma. Imagine uma lista que possui as seguintes opções: [1, 2, 3, 4]

Nesse exemplo, uso apenas um objeto, para cada opção escolhida, ele mostrará esse texto na grid, lembrando que como apresentação, ele não altera o valor:

```
var gridAPI = Form.grids ('GRID');
gridAPI.columns('COLUNA').format({
        "1": "Valor um",
        "2": "Valor dois",
        "3": "Valor três",
        "4": "Valor quatro"
});
```

O resultado seria o seguinte:

Existem outras formas para esse tipo de formatação. Nesse exemplo, pode-se passar uma função dinâmica para alguma opção, o retorno da função irá determinar o valor de apresentação da célula. Nessa função, entra um objeto com duas propriedades, **value** e

tableRowData, **value** é valor da célula, e **tableRowData** é o valor de todas as colunas. Lembrando que o valor não será alterado, somente a apresentação.

```
var gridAPI = Form.grids ('GRID');
gridAPI.columns('COLUNA').format({
        "1": function(data) {
                return data.tableRowData.LISTFORMATGRID + " valor um";
        },
        "2": function(data) {
                return data.tableRowData.LISTFORMATGRID + " valor dois";
        },
        "3": function(data) {
                return data.tableRowData.LISTFORMATGRID + " valor três";
        },
        "4": "valor quatro"
});
```

Nesse outro exemplo, passa-se uma função direto, e não uma por opção. **No caso de uma coluna dinâmica** (que foi adicionada via código), **essa opção seria a mais adequada**, pois essa coluna dinâmica, nunca teria um valor.

```
var gridAPI = Form.grids('GRID');
gridAPI.columns('COLUNA').format(function(data) {
        let valueList = typeof data.tableRowData.LISTFORMATGRID === 'object' &&
        data.tableRowData.LISTFORMATGRID.length ?
data.tableRowData.LISTFORMATGRID[0] : data.tableRowData.LISTFORMATGRID;
        switch(valueList) {
          case "1":
                return valueList + " valor um";
                break;
          case "2":
                return valueList + " valor dois";
                break;
          case "3":
                return valueList + " valor três";
                break;
          case "4":
                return "valor quatro";
                break;
          default:
                return "Default";
                break;
```

```
        }
});
```

Agora também é possível adicionar um botão na célula da grid, os nomes dos ícones disponíveis, segue a lista do material do google (https://material.io/icons/)
Para adicionar é da seguinte maneira:

```
var gridAPI = Form.grids ('GRID');
gridAPI.columns('COLUNA').format({
        component: {
                type: 'button',
                config: {
                        icon: 'link',
                        name: 'Download',
                        onClick: function() {
                                console.log("onClick");
                        }
    }
});
```

12. Funções de campos do tipo Autocomplete:

addOptions customizado – Adiciona novas opções ao campo de lista de acordo com o exemplo abaixo:

```
var autocompleteAPI = Form.fields('LISTA');

// Adicionando uma opção de valor para o campo com ID LISTA e limpando toda a lista, mantendo somente as opções adicionadas no momento
autocompleteAPI.addOptions(['Opção 1'], true).apply();

// Adicionando uma opção de valor para o campo com ID LISTA e limpando toda a lista, mantendo somente as opções adicionadas no momento e mantendo a opção selecionada anteriormente

autocompleteAPI.addOptions(['Opção 1'], true, true).apply();
```

addOptions – Adiciona novas opções ao campo de lista. Recebe como parâmetro as opções que devem ser adicionas. Esse parâmetro pode ter 3 (três) formatos:

- Valor do tipo **string** representando o valor de uma opção que será adicionada.

  Exemplo:

```
var autocompleteAPI = Form.fields('LISTA');
```

```
// Adicionando uma opção de valor para o campo com ID LISTA.
autocompleteAPI.addOptions('Opção 1').apply();
```

- Valor do tipo **array** e cada item desta lista deve ser do tipo string, representando os valores das opções que irão ser adicionas.

  Exemplo:

```
var autocompleteAPI = Form.fields('LISTA');

// Adicionando várias opções de valor para o campo com ID LISTA.
autocompleteAPI.addOptions([
        Opção 1',
        'Opção 2',
        'Opção 3'
]).apply();
```

- valor do tipo **array** e cada item da lista for do tipo **object** com duas propriedades: **name** para o nome a ser exibido no campo e **value** para o valor deste campo que será enviado na submissão do Form.

  Exemplo:

```
var autocompleteAPI = Form.fields('LISTA');

// Adicionando várias opções de valor para o campo com ID LISTA.
autocompleteAPI.addOptions([
        { name: 'Opção 1', value: 1 },
        { name: 'Opção 2', value: 2 },
        { name: 'Opção 3', value: 3 },
]).apply();
```

clear – Remove todas as opções do campo de lista.

Exemplo:

```
var autocompleteAPI = Form.fields('LISTA');

// Removendo todas as opções do campo com ID LISTA.
autocompleteAPI.clear().apply();
```

removeOptions – Remove opções do campo de lista. Recebe como parâmetro um valor do tipo **string** representando o valor da opção que deve ser removida, ou um **array** com uma lista de valores de opção que devem ser removidos.

Exemplo:

```
var autocompleteAPI = Form.fields('LISTA');

// Removendo uma opção do campo com ID LISTA.
autocompleteAPI.removeOptions('Opção 1').apply();

// Removendo várias opções do campo com ID LISTA.
autocompleteAPI.removeOptions(['Opção 2', 'Opção 3']).apply();

//Reset lista (Limpa lista).
Form.fields("LISTA").removeOptions([]).apply();
```

13. Ação de Form

Componente de ação direta do Form, usualmente representam as ações de aprovação e rejeição de etapa, o acesso ocorre através da função **actions** do componente **Form** da API.

Exemplo de acesso a um componente deste tipo:

```
// API para a ação de aprovação de etapa (ID aprovar).
var aprovarAPI = Form.actions('aprovar').execute();
```

14. Ação de Campo

Componente de ação relacionada a um campo básico do Form, usualmente representam as ações de refresh, lupa e ação de aplicação externa associada a um determinado campo.

Exemplo de acesso a um componente deste tipo:

```
// API para a ação de lupa do campo com ID NOME (ID NOME_lookup).
var nomeLupaAPI = Form.fields('NOME').actions('NOME_lookup');
```

15. Ação de Campo de Grid

Componente de ação relacionado a um campo de Grid.
Exemplo de acesso a um componente deste tipo:

```
// API para a ação de refresh do campo com ID PRECO do grid com ID GRID1 (ID PRECO_refresh).
var precoRefreshAPI = Form.grids('GRID1').fields('PRECO').actions('PRECO_refresh');
```

Propriedades:

confirmsWith – Valor do tipo **string** que define uma mensagem de confirmação que é exibida para o usuário antes da ação ser executada, que também permite o cancelamento desta execução.

Exemplo:

```
// Adiciona uma mensagem de confirmação para a ação de rejeição de etapa.
Form.actions('rejeitar').confirmsWith('Tem certeza que deseja rejeitar esta etapa?').apply();
```

disabled – Valor do tipo booleano que define se o botão da ação está desabilitado ou não.

Exemplo:

```
// Desabilita a ação de aprovação de etapa, se o valor do campo com ID PRECO for menor ou igual a 0.
var disableRejeitar = (Form.fields(PRECO).value() <= 0.0);

Form.actions('aprovar').disabled(disableRejeitar).apply();
```

hidden – Valor do tipo booleano que define se o botão da ação está escondido para o usuário ou não.

Exemplo:

```
// Oculta o botão da ação de rejeitar.
Form.actions('rejeitar').hidden(true).apply();
```

icon – Valor do tipo **string** que define um identificador de ícone para a ação. A referência de ícones e seus identificadores estão disponível na seção Dicas (**item 27**).

Exemplo:

```
// Define um novo ícone de histórico para a ação de refresh do campo com ID HIST.
Form.fields('HIST').actions('HIST_refresh').icon('history').apply();
```

label – Valor do tipo **string** que define o nome exibido no botão da ação.

Exemplo:

```
// Define o label 'Aceitar' para o botão da ação de aprovação.
Form.actions('aprovar').label('Aceitar').apply();
```

style – Valor do tipo **string** que define um identificador de estilo para a ação. Os valores possíveis para este campo são: **primary**, para a cor azul, de botões que

alteram e submetem valores no form, ou de aprovação de etapa; **cancel**, para a cor cinza de botões de aplicação externa e que cancelam ações de alteração; ou **danger** para a cor vermelha de botões que executam ações destrutivas ou de rejeição de etapa.

Exemplo:

```
// Define o estilo primary para a ação de lupa do campo com ID QUANTIDADE do grid com ID GRID.
Form.grids('GRID').fields('QUANTIDADE').actions('QUANTIDADE_lookup').style('primary').apply();
```

Funções:

execute – Executa qualquer ação via API, como aprovação, rejeição de etapa, etc.

Exemplo:

```
// Executa a ação de aprovação de etapa via API.
Form.actions('aprovar').execute();
```

16. Lendo propriedades dos componentes do Form:

Para acessar as propriedades de um componente do Form é simples e pode ser feita de duas maneiras.

Através do get:
Todo componente da API possui uma função chamada **get** para leitura de suas propriedades. Se ela for chamada sem parâmetro será retornado os valores de todas as propriedades do componente em um objeto, se for chamada passando o nome de uma propriedade como parâmetro, será retornado apenas o valor da propriedade informada.

Exemplo:

```
// Retornando um objeto com os valores de todas as propriedades do campo com ID NOME.
var propsCampoNome = Form.fields('NOME').get();

// Retornando o valor do campo com ID NOME (propriedade _value_).
var valorNome = Form.fields('NOME').get('value');
```

Através das funções das propriedades:
Para facilitar o acesso aos valores das propriedades, a API disponibiliza uma função com o nome de cada propriedade disponível de um determinado componente. Se ela for chamada sem parâmetro será retornado o valor da propriedade relacionada a função.

Exemplo:

```
// Retornando o valor do campo com ID NOME.
var valorNome = Form.fields('NOME').value();
```

17. Definindo propriedades para os componentes do Form:

Importante: Após executar o comando para definir valores em um componente, a função **apply()** deve ser chamada logo em seguida para que essas mudanças sejam refletidas na tela. Esta função pode ser chamada de qualquer componente da API, ou da própria variável global **formAPI.**

Definir novos valores para propriedades dos componentes do Form é simples e também pode ser feito de duas formas.

Através do set: Todo componente da API possui uma função chamada **set** para alteração de suas propriedades. Ela recebe como parâmetro um objeto javascript, onde suas chaves são os nomes das propriedades que devem ser alteradas e os valores são os novos valores que as propriedades terão. Esta função é útil para o caso onde é necessário alterar muitas propriedades de um componente de uma só vez.

Exemplo:

```
// ANTES era possível definir um novo valor e um novo label para o campo com ID NOME
Form.fields('NOME').set({ value: 'Fulano de Tal', label: 'Nome Completo' }).apply();

// AGORA também é possível definir um novo valor e um novo label para o campo da seguinte forma:
Form.fields('NOME').value('Fulado de tal').label('Nome Completo').apply();
```

Através das funções das propriedades: Assim como no caso da leitura, as funções com o nome das propriedades de um componente também podem ser utilizada para definir novos valores para elas, se ela for chamada com um parâmetro, representando o novo valor que será definido. Além disso estas funções podem ser encadeadas para a necessidade de alteração de mais de uma propriedade para um determinado componente.

Exemplo:

```
// Definindo um novo nome e ícone para a ação de aprovação
Form.actions('aprovar').label('Gravar').icon('save').apply();
```

18. Eventos do Form:

SUBMIT – Publicado quando um Form é submetido através de qualquer ação (Aprovação ou Rejeição de Etapa, por exemplo). Útil para incluir validações customizadas via API, pois a submissão do Form pode ser interrompida ao lançar uma exceção em uma função que assina este event.

Obs.: Caso houver algum tipo de inserção de erro e/ou validação do form a partir da APIJS, a responsabilidade de interromper o submit é da própria APIJS, invocando o reject() no escopo do subscribe, do contrário o form só irá ser interrompido segundo as regras de modelagem no studio.

Exemplo:

```
// No evento de submissão do Form, realizar validação dos campos CPF e CEP.
Form.subscribe('SUBMIT', function(formId, actionId, reject) {
	var cpfAPI = Form.fields('CPF');
	var cepAPI = Form.fields('CEP');
	var formInvalido = false;

	if (!validaCPF(cpfAPI.value())) {
		cpfAPI.errors('CPF Inválido');
		formInvalido = true;
	}
	if (!validaCEP(cepAPI.value())) {
		cepAPI.errors('CEP Inválido');
		formInvalido = true;
	}
	if (formInvalido) {
		 reject();
	}
});
```

SUBMIT_SUCCESS – Publicado quando a submissão do Form é completada com sucesso. Útil para realizar algum redirecionamento do usuário quando o mesmo completa uma etapa.
SUBMIT_ERROR – Publicado quando ocorre um erro na submissão do Form que não é relacionado com validação de campos.
VALIDATION_ERROR – Publicado quando ocorre um erro de validação dos campos do Form, quando o mesmo é submetido.

19. Eventos de Campos:

BLUR – Publicado quando o campo perde o foco.

FOCUS – Publicado quando o campo ganha foco.

CLICK – Publicado quando o campo é clicado.

CHANGE – Publicado quando algum valor via teclado é inserido no campo.

KEY_PRESS – Publicado a cada iteração com o campo.

SET_FIELD_VALUE – Publicado sempre que o "value" do campo é alterado pela API JS. Não é disparado quando digitado diretamente no campo. Para isso use os eventos CHANGE ou KEY_PRESS.

Motivo: O motivo de criar esse novo evento é que o evento **SET_FIELD_PROPERTIES** pode causar lentidão no formulário quando utilizado, pois ele é disparado quando há uma alteração em qualquer propriedade de um elemento e não apenas na propriedade "value".

Obs.: O vento **SET_FIELD_PROPERTIES** não foi removido do sistema, porém recomendamos que conforme ocorrer manutenção nos JS de clientes o evento seja substituído pelo evento **SET_FIELD_VALUE**.

Exemplo:

```
// A alteração de valor do campo com ID IDADE implica no recálculo do campo com ID
PREMIO_SEGURO
Form.fields('IDADE').subscribe('SET_FIELD_VALUE', () => {
        Form.fields('PREMIO_SEGURO').value(calculaPremioSeguro()).apply();
});

//Exemplo (IE) – Internet Explorer
Form.fields('IDADE').subscribe('SET_FIELD_VALUE', function() {
        Form.fields('PREMIO_SEGURO').value(calculaPremioSeguro()).apply();
});
```

20. Eventos do Grid:

GRID_SUBMIT – Publicado quando uma linha é adicionada ou atualizada em um Grid. Útil para realizar o recálculo de campos totalizadores usados para exibir agregações de valores do Grid (média ou soma). Ou para realizar a validação dos valores entrados no Grid, da mesma forma que está exemplificado no evento **SUBMIT** do Form.
GRID_ADD_SUBMIT – Publicado apenas quando uma linha é adicionada em um Grid.
GRID_EDIT_SUBMIT – Publicado apenas quando uma linha é atualizada em um Grid.

GRID_EDIT – Publicado quando o usuário seleciona uma linha de um Grid para edição de seus valores. Útil para preparar algum campo com os valores da linha a ser editada.
GRID_DESTROY – Publicado quando uma linha é removida do Grid. Útil para realizar o recálculo de campos totalizadores da mesma forma que **GRID_SUBMIT**
GRID_RESET – Publicado quando o usuário cancela a edição dos valores de uma linha de um Grid.

```
Form.grids("ID_GRID").subscribe("GRID_ADD_SUBMIT", => {
      //ação a ser realizada
});
```

21. Eventos de Ações

CLICK – Publicado quando o botão da ação é clicado.

22. Eventos de Ações do tipo SUBMIT (ações de mudança de Etapa)

SUBMIT – Publicado quando o Form é submetido através da ação específica. Útil para o caso de realizar validações apenas para um determinado tipo de ação (aprovar, rejeitar, cancelar ou finalizar Etapa).
SUBMIT_SUCCESS – Publicado quando a submissão do Form é completada com sucesso através da ação específica. Útil para realizar algum redirecionamento do usuário apenas na aprovação da etapa, por exemplo.
SUBMIT_ERROR – Publicado quando ocorre um erro na submissão do Form, que não é relacionado com validação de campos, para a ação específica.
VALIDATION_ERROR – Publicado quando ocorrem um ou mais erros na validação dos campos do Form, quando o mesmo é submetido pela ação específica.

23. Eventos de Ações do tipo SEARCHER (Refresh e Lupa)

Eventos genéricos para modais:

SEARCHER_LOAD_SUCCESS – Publicado quando o carregamento do resultado da Lupa ou do valor do Refresh é realizado com sucesso.
SEARCHER_LOAD_ERROR – Publicado quando ocorrem um erro durante o carregamento do resultado da Lupa ou do valor Refresh.

Eventos para componente Lupa específico

SEARCHER_SELECT – Publicado quando um item da **Lupa** é selecionado de acordo com um componente específico.

Exemplo:

```
Form.fields("LUPA1").subscribe("SEARCHER_SELECT", function (itemId, inputId, response) {
	console.log(response);
});
```

24. Dicas

Listar Ids dos componentes do Form: Uma forma prática para listar os Ids de campos e ações ligados a um Form, normalmente em console devTools do Chrome para testar scripts antes de integrá-los para uso no Studio, é utilizando a função javascript **map()**, que transforma valores de arrays:

Exemplos:

```
// Listando os IDs de todos os campos do Form:
// Ex. de retorno: ["LINHADETEXTO", "CAIXA", "INTEIRO", "NUM_DEC", "GRID", "LISTA",
"BTN_GRAFICO"]

Form.fields().map(f => f.id);

// Listando os IDs para todas as ações de um Form:
// Ex. de retorno: ["aprovar", "cancel"]
Form.actions().map(a => a.id);

// Listando os IDs de todos os campos de um Grid:
// Ex. de retorno: ["NOME_ITEM", "PRECO_UNITARIO", “QUANTIDADE”]
Form.grids('GRID').fields().map(f => f.id);

// Listando os IDs de todas as ações de um campo:
// Ex. de retorno: ["CEP_lookup"]
Form.fields('CEP').actions().map(a => a.id);

// Exemplo (IE) – Internet Explorer
Form.fields().map(function(f){
	return f.id
});

// Listando os IDs para todas as ações de um Form:
// Ex. de retorno: ["aprovar", "cancel"]
Form.actions().map(function(a){ return a.id});

// Listando os IDs de todos os campos de um Grid:
// Ex. de retorno: ["NOME_ITEM", "PRECO_UNITARIO", “QUANTIDADE”]
Form.grids('GRID').fields().map(function(f){ return f.id});

// Listando os IDs de todas as ações de um campo:
// Ex. de retorno: ["CEP_lookup"]
```

```
Form.fields('CEP').actions().map(function(a){ return  a.id});
```

Chamar o apply() no fim do seu script para aplicar as mudanças no Form apenas uma vez: A função **apply()**, da API, atualiza em tela todas as alterações aplicadas ao Form. Esta ação é separada nesta função para possibilitar que mais de uma alteração realizada via API possam ser feitas em apenas 1 processo de renderização da tela, ocasionando num ganho de performance do Form e em uma melhora na experiência do usuário.

Exemplo:

```
// Forma incorreta, 6x o processo de renderização é executado.
Form.fields('NOME').value('Teste1234').disabled(true).apply();
Form.fields('DATA_NASCIMENTO').value(new Date()).apply();
Form.fields('LISTA_DE_ESTADOS').value('RJ').apply();
Form.fields('CPF').value('11111111111').readOnly(true).apply();
Form.fields('RG').value('22222222222').apply();
Form.fields('OBSERVACAO').value('Teste da API-JS').apply();


// Forma correta, o processo de renderização é executado somente uma vez.
Form.fields('NOME').value('Teste1234').disabled(true);
Form.fields('DATA_NASCIMENTO').value(new Date());
Form.fields('LISTA_DE_ESTADOS').value('RJ');
Form.fields('CPF').value('11111111111').readOnly(true);
Form.fields('RG').value('22222222222');
Form.fields('OBSERVACAO').value('Teste da API-JS');

// Aplica todas as modificações de uma só vez.
Form.apply();
```

Promise retornada pelo apply(): A execução da função **apply()** retorna uma **Promise**, que pode ser utilizada para lidar com operações assíncronas de forma que operações subsequentes possam ser executadas após a finalização da operação assíncrona. No caso da API-JS a execução da função **apply()** é uma operação assíncrona e em alguns momentos durante a utilização da API-JS pode surgir a necessidade de executar uma operação após a renderização em tela do resultado da operação anterior, desta forma pode ser utilizado o método **then()** da **Promise** retornada pela primeira operação executada com o método **apply()** e assim encadear operações.

Exemplo:

```
var gridAPI = Form.grids('GRID');
gridAPI.fields('NOME').value('Teste');
gridAPI.fields('SOBRENOME').value('API-JS');
```

```
var promise =Form.apply();

// Execução da operação abaixo só irá ocorrer após a renderização em tela das operações acima, sendo assim quando isso ocorrer ira ser disparada a ação de submit do grid, obtendo os valores dos campos modificados acima.
promise.then(() => Form.actions('GRID_SUBMIT').execute().apply());
```

É possível encadear mais de uma ação.

Exemplo:

```
var gridAPI = Form.grids('GRID');
gridAPI.fields('NOME').value('Teste');
gridAPI.fields('SOBRENOME').value('API-JS');

var promise1 = Form.apply();
var promise2 = promise1.then(() => new Promise((resolve, reject) => {
        Form.actions('GRID_SUBMIT').execute().apply().then(resolve);
});

promise2.then(() => console.log('Todas as operações foram finalizadas'));
```

Referência: **Promise**

É possível encadear ações no campo.
Exemplo:

```
var formField = Form.fiels('ID_CAMPO);
formField.value('Valor do campo').label('Label do campo').apply();
```

25. Máscaras pré-definidas para campos

É possível definir máscaras para os campos, como foi citado em outras seções da documentação, existem algumas máscaras pré-definidas para uso, segue uma lista das mesmas e como utilizá-las.

1. **cpf** – XXX.XXX.XXX-XX

```
Form.fields('NOME').mask('cpf').apply();
```

2. **cnpj** – XX.XXX.XXX/XXXX-XX

```
Form.fields('NOME').mask('cnpj').apply();
```

26. Máscaras customizadas para campos

É possível definir máscaras customizadas também, caso as máscaras predefinidas não atendam algum cenário específico. A biblioteca que é utilizada pela aplicação para trabalhar com máscaras em campos é a **jQuery Mask Plugin (Link da documentação).** Desta forma toda a lógica de customização de máscara é definida pela mesma, sendo que na documentação da biblioteca se encontram outros exemplos de como definir máscaras customizadas também.

```
Form.fields('NOME').mask('00/00/0000').apply();
```

27. Opções para ícones de ações

Os ícones disponíveis para utilização no form são os que estão suportados pelo material design do Google. Para utiliza-los basta escolher o ícone, obter o identificador dele e definir na propriedade icon pela API. Segue abaixo o site com uma lista para todos os ícones disponíveis.

Site: https://material.io/icons/

28. Funções para validações comuns

É possível definir validações para os campos do form, basta utilizar o método **validations()** da API, segue um exemplo abaixo:

Campo obrigatório – A validação de campo obrigatório no Form pode estar atrelada a ações do Form, sendo essas a de aprovar, cancelar, rejeitar e finalizar. O atributo **required** é utilizado para definir a obrigatoriedade através de um valor booleano, sendo esse **true** para obrigatório e **false** para opcional.

1. **Aprovar**

```
// Torna o campo teste obrigatório para a ação de aprovar.

Form.fields('TESTE').setRequired('aprovar', true).apply();
```

2. **Cancelar**

```
// Torna o campo teste obrigatório para a ação de cancelar.
Form.fields('TESTE').setRequired('cancel', true).apply();
```

3. **Rejeitar**

```
// Torna o campo teste obrigatório para a ação de rejeitar.
Form.fields('TESTE').setRequired('rejeitar', true).apply();
```

4. Finalizar

```
// Torna o campo teste obrigatório para a ação de finalizar.
Form.fields('TESTE').setRequired('finish', true)
```

5. Botão de adicionar do Grid

```
// Torna o campo nome do grid obrigatório para a ação do botão de adicionar do grid.
Form.grids.('ID_GRID')fields('TESTE').setRequired(true);
```

6. Tornar campo opcional

```
Form.fields('TESTE').setRequired('aprovar', false)
```

7. **Definir múltiplas validações em um campo**

```
Form.fields('TESTE').validations([{formActionName: 'aprovar', required: true}, {formActionName:
'cancel', required: true}]).apply();
```

29. Tonar inserção de itens no grid obrigatório

A definição desse comportamento pode ser feita pela API-JS, através do método subscribe() da api, nas ações do form ou para todas as açoes de SUBMIT do mesmo, segue alguns exemplos abaixo:

Todas as ações de SUBMIT no form.

```
Form.subscribe('SUBMIT', function(formId, actionId, reject) {
        if (formAPI.fields('ITEMS').dataRows().length === 0) {
                reject();
        }
});
```

Escopado por Ação (Aprovar, Rejeitar, ...) do Form.

```
Form.subscribe('SUBMIT', function(formId, actionId, reject) {
        if (formAPI.fields('ITEMS').dataRows().length === 0) {
                reject();
        }
```

```
});
```

30. Implementação de funcionalidade para validação de documento no Template

Importante: Essa validação só será realizada através da importação do documento.

Exemplo:

```
Form.fields("TEMPLATE").subscribe("DOCUMENT_IMPORT_FILE_SUBMITTED", function (storeId, inputId, file) {
        console.log(file);
});
```

A "validação" do documento acontece no escopo do subscribe, que retorna o documento importado e suas propriedades.

31. Implementação de mensagem na modal de documento no Template

A definição desse comportamento pode ser feita pela API-JS, através do método addMessageModal() para o campo do tipo "TEMPLATE" do form, segue alguns exemplos abaixo:

```
var field = Form.fields("ID_CAMPO");

// error – mensagem de erro
field.addMessageModal({
        message: "mensagem a ser apresentada", type: 'error'
});

// warning – mensagem de alerta
field.addMessageModal({
        message: "mensagem a ser apresentada", type: 'warningr'
});

// info – mensagem de informação
field.addMessageModal({
        message: "mensagem a ser apresentada", type: 'info'
});
```

32. Criando mensagem de aviso em modal

A definição desse comportamento pode ser utilizada através da API-JS, com o método addCustomModal (), pode ser informado o título da modal, a mensagem a ser apresentada, nome, ação e ícone do botão, conforme o exemplo abaixo:

```
Form.addCustomModal({
        title: "Título modal",
        description: "mensagem a ser apresentada",
       //false para não aparecer o botão “fechar” padrão da modal
        showButtonClose: false,
        buttons: [{
                name: "Confirmar",
                icon: "add",
                // true: para fechar a modal após a ação ser realizada e false: para não fechar a modal
                closeOnClick: true,
                action: function () {
                        //ação a ser realizada
                }
        }]
});
```

Obs.: Caso não seja passado nenhum valor para os botões a ação dos mesmos serão padrões.

33. Criando loading através da API-JS

A definição desse comportamento pode ser feita pela API-JS, através do método showLoader() pode ser informado a mensagem a ser apresentada e ícone de loading, conforme o exemplo abaixo:

```
// description – descrição da mensagem a ser apresentada
// icon – true/false
Form.showLoader({
        description: "mensagem a ser apresentada",
        icon: true
});
```

Para finalizar a ação de loading pode ser utilizado o método hideLoader() conforme o exemplo abaixo:

```
// fecha o loader
Form.hideLoader();
```

OBS.: Para realizar o fechamento da modal de loading é necessário utilizar o método **hideLoader().**

34. Pegando informações da atividade

É possível pegar informações da atividade como o código da etapa, ciclo, título, etc. Para isso basta utilizar: **ProcessData** conforme os exemplos abaixo:

```
// Pegar o título da etapa:
ProcessData.activityTitle;

// Pegar o ciclo da etapa:
ProcessData.cycle;

// Pegar o código da etapa:
ProcessData.activityInstanceId;
```

35. Boas prática

Segue abaixo algumas instruções para melhorar a experiência de uso e a performance da APIJS:

1. **Uso do subscribe SET_FIELD_VALUE**

O subscribe SET_FIELD_VALUE é publicado sempre o “value” do campo é alterado pela APIJS. Porém não é disparado quando digitado diretamente no campo.

2. **Uso do subscribe CHANGE**

O subscribe CHANGE é disparado quando houver qualquer alteração no **valor** do campo referenciado, ou seja, quando um novo valor é inserido no campo.

Obs.: O .value() do campo referenciado ainda não vai estar atualizado na chamada do subscribe.

3. **Encapsulamento do script**

Correto:

```
(function () {
	var form;

	$(document).ready(function (){
		//código aqui
	});
```

```
})();
```

4. **Mudança de propriedade de um componente**

   Correto:

```
var fieldAPI =  Form.fields('CAMPO')

fieldAPI.value('valor1').apply();
fieldAPI.value('valor2').apply();
fieldAPI.value('valor3').apply();
```

5. **Outras práticas a serem seguidas**

   Evitar o uso de promises (.then);
   Fazer a validação dos campos somente no submit do form, e não em tempo real;
   Adotar somente a utilização de um método de inserção de erros, ou por meio do form, ou por meio dos campos;
   Evitar o uso de seletores JQuery que não fazem parte da APIJS;
   Utilizar somente a APIJS para a manipulação de qualquer estado/estilo dos campos.

36. Ouvindo ação de desvincular documento de um campo template

Exemplo:

```
Form.fields("TEMPLATE").subscribe("UNLINKED_FILE_SUCCESS", function (formId, fieldId) {
        console.log(formId, fieldId);
});
```

37. Ouvindo ação de excluir documento de um campo template

Exemplo:

```
Form.fields("TEMPLATE").subscribe("DELETE_FILE_SUCCESS", function (formId, fieldId) {
        console.log(formId, fieldId);
});
```

38. Ouvindo ação de save do form

Exemplo:

```
Form.subscribe("SAVE_FORM_SUCCESS", function (formId) {
        console.log("Salvou form com sucesso");
});
```

39. Alterando as propriedades de uma ou mais linhas de uma grid

Escopo do método:

```
Form.grids("GRID1").updatePropsRows(function | Integer | null, Object).apply();
```

Parâmetros do método:

1. Função, **Número inteiro** ou **null**:

Função – Função de filtro, os dados (linhas) que entrarem na filtragem, serão afetados.
Número inteiro – O dado (linha) da grid que estiver no índice do array será afetado.
null – Todos os dados (linhas) serão afetados.

2. Objeto { **visibleRow**: true | false, **visibleEdit**: true | false, **visibleDelete**: true | false, **visibleSigned**: true | false }

visibleRow – **true** para deixar visível as linhas filtradas, ou **false** para ocultá-las.
VisibleEdit – **true** para deixar o botão editar da linha visível, ou **false** para escondê-lo.
VisibleDelete – **true** para deixar o botão deletar da linha visível, ou **false** para escondê-lo.
VisibleSigned – **true** para deixar o botão do assinador da linha visível, ou **false** para escondê-lo.

Exemplos:

1. No exemplo abaixo, todas as linhas que tiverem a coluna “L1” com o texto igual a “teste”, serão afetadas, na grid “GRID1”. A linha toda será oculta (visibleRow), o botão de editar (visibleEdit) e o botão de deletar (visibleDelete) também ficarão ocultos:

```
Form.grids("GRID1").updatePropsRows(function (dataRow) {
        return daraRow["L1"] === "teste";
}, {visibleRow: false,visibleEdit: false,visibleDelete: false}).apply();
```

2. No exemplo abaixo, somente a primeira linha (0) da grid "GRID1" será afetada. A linha toda será oculta (visibleRow), o botão de editar (visibleEdit) e o botão de deletar (visibleDelete) também ficarão ocultos:

```
Form.grids("GRID1").updatePropsRows(0, {visibleRow: false,visibleEdit: false,visibleDelete:
false}).apply();
```

3. No exemplo abaixo, todas as linhas da grid "GRID1" serão afetadas. A linha toda ficará visível (visibleRow), o botão de editar (visibleEdit) e o botão de deletar (visibleDelete) ficarão ocultos:

```
Form.grids("GRID1").updatePropsRows(null, {visibleRow: true,visibleEdit: false,visibleDelete:
false}).apply();
```

4**.** No exemplo abaixo, todas as linhas da grid "GRID1" serão afetadas. O botão do assinador (visibleSigned) ficará oculto:

```
Form.grids("GRID1").updatePropsRows(null, {visibleSigned: false}).apply();
```

OBS.: Se uma ou mais linhas forem ocultas pelo método updatePropsRows, as informações de paginação da grid (total de itens, número de páginas, etc) não serão afetadas, pois quem lida com essas propriedades é o back-end.

40. Eventos adicionados na 550_v1.04

GRID_DATA_ROW_LOADED

Evento disparado quando os dados da grid terminarem de carregar e o dataRows estiver disponível via apijs, no carregamento do form. O intuito desse evento é evitar setTimeout nos scripts, a fim de esperar o dataRows ficar disponível para ser usado.

Parâmetros do evento:

- **formId** - Id do form;
- **gridId** - Id da grid;
- **dataRows** - dados da grid.

```
Form.grids("G1").subscribe("GRID_DATA_ROW_LOADED", function (formId, gridId, dataRows) {
  //usar o dataRows aqui sem a necessidade de setTimeout
});
```

GRID_ADD_BEFORE

Evento disparado antes dos dados serem inseridos na grid pelo usuário. O intuito desse evento é ter a possibilidade de cancelar a inserção dos dados na grid.

Parâmetros do evento:

- **formId** - Id do form;
- **gridId** - Id da grid;
- **submittedDataRow** - dados que serão inseridos;
- **dataRows** - dados da grid antes da inserção;
- **resolve** - função para permitir a inserção dos dados;
- **reject** - função para rejeitar a inserção dos dados.

```
Form.grids("G1").subscribe("GRID_ADD_BEFORE", function (formId, gridId, submittedDataRow, dataRows, resolve, reject) {
  //vai entrar aqui quando o usuario clicar para inserir os dados na grid G1. Os dados serao inseridos normalmente.
});

Form.grids("G1").subscribe("GRID_ADD_BEFORE", function (formId, gridId, submittedDataRow, dataRows, resolve, reject) {
  //vai entrar aqui quando o usuario clicar para inserir os dados na grid G1, porem os dados nao serao inseridos, pois o reject foi usado. A mensagem "erro ao inserir os dados" sera exibida na grid
  reject("erro ao inserir os dados");
});

Form.grids("G1").subscribe("GRID_ADD_BEFORE", function (formId, gridId, submittedDataRow, dataRows, resolve, reject) {
  //vai entrar aqui quando o usuario clicar para inserir os dados na grid G1. Os dados serao inseridos normalmente.
  resolve();
});

Form.grids("G1").subscribe("GRID_ADD_BEFORE", function (formId, gridId, submittedDataRow, dataRows, resolve, reject) {
  //vai entrar aqui quando o usuario clicar para inserir os dados na grid G1. Os dados serao adicionados normalmente, porem o campo TXT sera inserido com o valor "teste"
  submittedDataRow.TXT = "teste";
  resolve(submittedDataRow);
});
```

GRID_ADD_AFTER

Evento disparado depois dos dados serem inseridos na grid pelo usuário. O intuito desse evento é de garantir que o dataRows terá o valor atualizado após a inserção.

Parâmetros do evento:

- **formId** - Id do form;
- **gridId** - Id da grid;
- **submittedDataRow** - dados que foram inseridos;
- **dataRows** - dados atualizados da grid.

```
Form.grids("G1").subscribe("GRID_ADD_AFTER", function (formId, gridId, submittedDataRow, dataRows) {
  //vai entrar aqui depois que os dados forem inseridos na grid G1.
});
```

GRID_EDIT_BEFORE

Evento disparado antes dos dados serem editados na grid pelo usuário. O intuito desse evento é ter a possibilidade de cancelar a edição dos dados na grid.

Parâmetros do evento:

- **formId** - Id do form;
- **gridId** - Id da grid;
- **submittedDataRow** - dados que serão editados;
- **dataRows** - dados da grid antes da edição;
- **resolve** - função para permitir a edição dos dados;
- **reject** - função para rejeitar a edição dos dados.

```
Form.grids("G1").subscribe("GRID_EDIT_BEFORE", function (formId, gridId, submittedDataRow, dataRows, resolve, reject) {
  //vai entrar aqui quando o usuario clicar para editar algum dado na grid G1. O dado sera editado normalmente.
});

Form.grids("G1").subscribe("GRID_EDIT_BEFORE", function (formId, gridId, submittedDataRow, dataRows, resolve, reject) {
  //vai entrar aqui quando o usuario clicar para editar algum dado na grid G1, porem o dado nao sera editado, pois o reject foi usado. A mensagem "erro ao editar o dado" sera exibida na grid
  reject("erro ao editar o dado");
});

Form.grids("G1").subscribe("GRID_EDIT_BEFORE", function (formId, gridId, submittedDataRow, dataRows, resolve, reject) {
  //vai entrar aqui quando o usuario clicar para editar algum dado na grid G1. O dado sera editado normalmente.
  resolve();
});

Form.grids("G1").subscribe("GRID_EDIT_BEFORE", function (formId, gridId, submittedDataRow, dataRows, resolve, reject) {
  //vai entrar aqui quando o usuario clicar para editar algum dado na grid G1. O dado serao editado normalmente, porem o campo TXT sera editado com o valor "teste"
```

```
 submittedDataRow.TXT = "teste";
 resolve(submittedDataRow);
});
```

GRID_EDIT_AFTER

Evento disparado depois dos dados serem editados na grid **G1** pelo usuário. O intuito desse evento é garantir que o dataRows terá o valor atualizado após a edição.

Parâmetros do evento:

- **formId** - Id do form;
- **gridId** - Id da grid;
- **submittedDataRow** - dados que foram editados;
- **dataRows** - dados atualizados da grid.

```
Form.grids("G1").subscribe("GRID_EDIT_AFTER", function (formId, gridId, submittedDataRow,
dataRows) {
 //vai entrar aqui depois que os dados forem editados na grid G1.
});
```

GRID_DELETE_BEFORE

Evento disparado antes dos dados serem excluídos na grid pelo usuário. O intuito desse evento é ter a possibilidade de cancelar a exclusão dos dados na grid.

Parâmetros do evento:

- **formId** - Id do form;
- **gridId** - Id da grid;
- **deletedDataRow** - dados que serão excluídos;
- **dataRows** - dados da grid antes da exclusão;
- **resolve** - função para permitir a exclusão dos dados;
- **reject** - função para rejeitar a exclusão dos dados.

```
Form.grids("G1").subscribe("GRID_DELETE_BEFORE", function (formId, gridId, deletedDataRow,
dataRows, resolve, reject) {
 //vai entrar aqui quando o usuario clicar para excluir algum dado na grid G1. O dado sera excluido
normalmente.
});

Form.grids("G1").subscribe("GRID_DELETE_BEFORE", function (formId, gridId, deletedDataRow,
dataRows, resolve, reject) {
 //vai entrar aqui quando o usuario clicar para excluir algum dado na grid G1, porem o dado nao sera
excluido, pois o reject foi usado. A mensagem "erro ao excluir o dado" sera exibida na grid
 reject("erro ao excluir o dado");
});
```

```
Form.grids("G1").subscribe("GRID_DELETE_BEFORE", function (formId, gridId, deletedDataRow, dataRows, resolve, reject) {
  //vai entrar aqui quando o usuario clicar para excluir algum dado na grid G1. O dado sera excluido normalmente.
  resolve();
});
```

GRID_DELETE_AFTER

Evento disparado depois dos dados serem excluídos na grid pelo usuário. O intuito desse evento é garantir que o dataRows terá o valor atualizado após a exclusão.

Parâmetros do evento:

- **formId** - Id do form;
- **gridId** - Id da grid;
- **deletedDataRow** - dados que foram excluídos;
- **dataRows** - dados atualizados da grid.

```
Form.grids("G1").subscribe("GRID_DELETE_AFTER", function (formId, gridId, deletedDataRow, dataRows) {
  //vai entrar aqui depois que os dados forem excluidos na grid G1.
});
```