Criando blocos cliente customizados (mobile e web)

O Cronapp permite criar novos blocos de programação (cliente ou servidor) e adicionar na lista de blocos do Editor de Blockly. Dessa forma, é possível criar funções para atender as necessidades de cada projeto.

Esse documento apresenta como utilizar o Javascript para criar seus próprios blocos de programação e desenvolver blocos mais complexos, como a personalização de parâmetros e *callback*, por exemplo. Caso necessite de blocos básicos, com a entrada de parâmetros e retorno, desenvolva-os de forma low-code, acesse o tópico "Criar blocos low-code customizados" da documentação Bloco de programação.

Nesta página

Pré-requisitos

Antes de começar a seguir os passos do tutorial é preciso ter certeza de que se tem um ambiente minimamente preparado para reproduzir o exemplo. Abaixo estão os requisitos principais.

Requisitos:

1. Projeto que possua o lado cliente (web ou mobile) criado. Caso haja dúvidas de como criar esse tipo de projeto acesse o link (Criar projeto).
2. Habilitar o botão Modo Avançado.
3. Possuir algum conhecimento em desenvolvimento front-end (High-code).

Visão Geral

Nesse tutorial iremos criar um bloco de programação em uma nova categoria para ser usado no editor de bloco de programação cliente. Apesar de apresentamos um exemplo para a aplicação web, ele também poderia ser utilizado em uma aplicação mobile sem apresentar incompatibilidade.

Localização dos blocos nativos

O código dos blocos de programação nativo do lado cliente (web e mobile) ficam dentro do diretório `node_module` de cada aplicação. Esse diretório é utilizado pelo Cronapp para armazenar diversas bibliotecas necessárias apenas durante o período de desenvolvimento do seu projeto. Por isso, não recomendamos realizar alterações manuais nesse diretório, pois essas alterações serão sobrescritas nas próximas atualização do Cronapp. Utilize esse conteúdo apenas como referências para a criação de novas funções.

Funções JavaScript dos blocos de programação nativos:

- **Web**: `src/main/webapp/node_modules/cronapi-js/cronapi.js`
- **Mobile**: `src/main/mobileapp/www/node_modules/cronapi-js/cronapi.js`

Importar blocos de exemplo

Os passos apresentados no tópico Criar funções e categorias e diversos recursos do tópico Estrutura do código foram compactados no arquivo abaixo, sendo possível importá-los diretamente em seu projeto Cronapp.

Arquivo 1 - Blocos de programação criados nessa documentação

Passos:

1. Baixe o arquivo novosBlocos.zip em seu computador.
2. Com o Cronapp em Modo Avançado, acesse o menu de contexto do diretório `js` da aplicação web (Endereço: `src/main/webapp/js/`) e selecione **Novo > Fazer Upload de um zip**.
3. Selecione o arquivo `novosBlocos.zip` do seu computador na janela **Envio de arquivo** no Cronapp e clique em **OK**.
4. Acesse o menu de contexto do arquivo `nomeArquivo.cronapi.js` que importamos do zip (Endereço: `src/main/webapp/js/novosBlocos/nomeArquivo.cronapi.js`) e selecione a opção **Importar dependência** (similar ao que foi feito na Figura 1.4).
5. Acesse o menu de contexto da raiz do projeto e selecione **Recompilar e Reabrir Projeto** (Figura 1.5).
6. Abra o editor de Bloco de programação web e verifique os novos blocos na categoria **Minhas Funções**.

Criar funções e categorias

Recomendamos que os arquivos JavaScript criados fiquem sempre dentro dos diretórios `js` nas aplicações mobile e web, inclusive dentro de um diretório específico.

Diretório recomendado para a criação de arquivos JavaScript:

- **Web**: `src/main/webapp/js/`
- **Mobile**: `src/main/mobileapp/www/js/`

Vamos criar um diretório chamado `novosBlocos` que irá conter o arquivo JavaScript referente ao novo bloco cliente. Clique com o botão direito sobre o diretório `js` (Endereço: `src/main/webapp/js/`) e selecione **Novo > Diretório**, insira o valor "novosBlocos" na janela Confirmação e clique em **OK**.

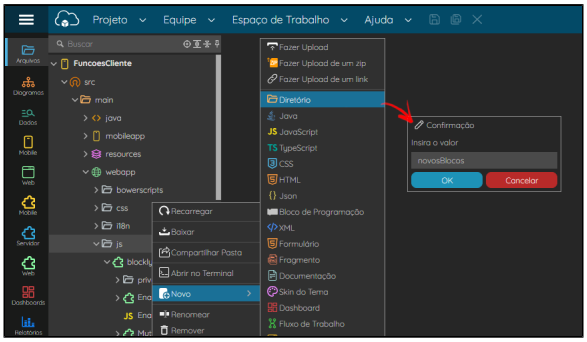

Figura 1 - Criando diretório

Agora vamos criar um arquivo JavaScript dentro do diretório criado. Clique com o botão direito no diretório `novosBlocos` (Endereço: `src/main/webapp/js/novosBlocos/`), selecione **Novo > JavaScript** (Figura 1.1).

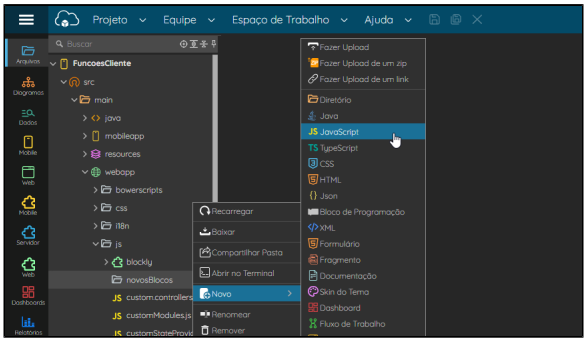

Figura 1.1 - Criando arquivo JavaScript

A janela de seleção de **Modelo** será exibida, permitindo criar um arquivo JavaScript vazio ou a partir de uma estrutura inicial. Selecione **Nova função cliente para Bloco de Programação** e clique em **Avançar**.

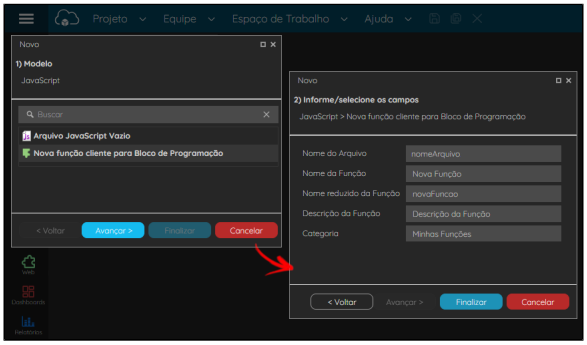

Figura 1.2 - Escolhendo o modelo do arquivo

Preencha os campos descritos e depois clique em **Finalizar**.

- **Nome do Arquivo:** nome do arquivo JavaScript que será criado dentro do diretório, a extensão do arquivo será "`*.cronapi.js`". Exemplo: `nomeArquivo.cronapi.js`
- **Nome da Função:** nome do bloco de programação, esse campo será utilizado na anotação `@name`.
- **Nome Reduzido da Função:** nome da função JavaScript:

Nome da função gerada por esse campo
`this.cronapi.myfunctions.nomeReduzido = function(){};`

- **Descrição da Função:** texto descritivo que será exibido na aba lateral da lista de blocos ou ao colocar o mouse em cima do bloco. Esse campo será utilizado na anotação `@description`.
- **Categoria:** nome da categoria que será criada para o bloco, esse campo será utilizado na anotação `@categoryName`.

  É importante não utilizar nomes de categorias existentes no editor de blockly.

Aparecerá uma janela de **Conflitos** informando que o arquivo `index.html` foi modificado. Para que a aplicação reconheça o conteúdo que estamos criando, é necessário que o arquivo JavaScript seja importando dentro `index.html`, por isso o Cronapp adiciona automaticamente. Clique em **Aplicar alterações aos itens marcados** para aceitar as alterações.

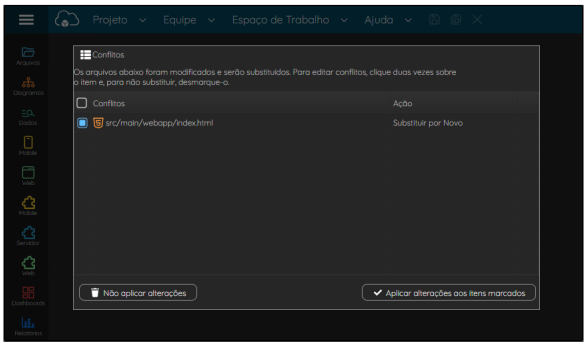

Figura 1.3 - Aplicando alterações no index.html

Caso queira remover a importação do arquivo no `index.html` (destaque 1 da figura 1.4), basta acessar o menu de contexto do arquivo JavaScript e selecionar **Remover dependência**, assim a aplicação não reconhecerá mais as chamadas dos novos blocos de programação.

Para adicionar novamente, acessar o menu de contexto do arquivo JavaScript e selecione **Importar dependência**.

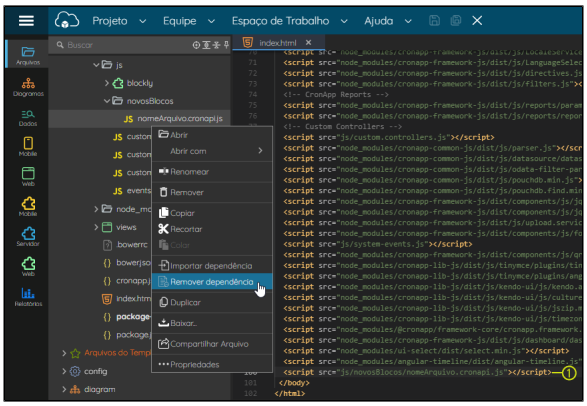

Figura 1.4 - Importação do arquivo JavaScript no index.html

Confirmada a alteração do conflito (figura 1.3), é necessário reiniciar o projeto para que o editor de Blockly reconheça o novo conteúdo. Acesse o menu de contexto da raiz do projeto e selecione **Recompil ar e Reabrir Projeto** (Figura 1.5).

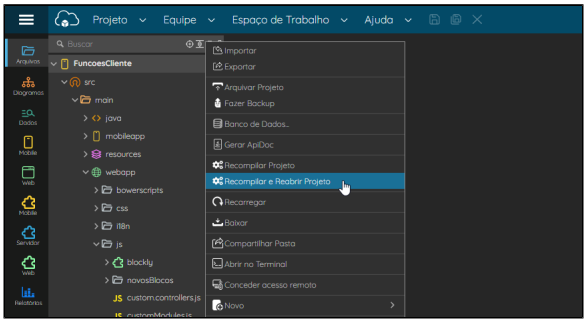

Figura 1.5 - Reinicializando o projeto

Por fim, abra um editor de bloco de programação cliente web para visualizar o bloco criado com os dados informados na etapa da figura 1.2. Como não fizemos nenhuma alteração na função JavaScript, o bloco executará uma ação genérica, retornando o valor passado por parâmetro.

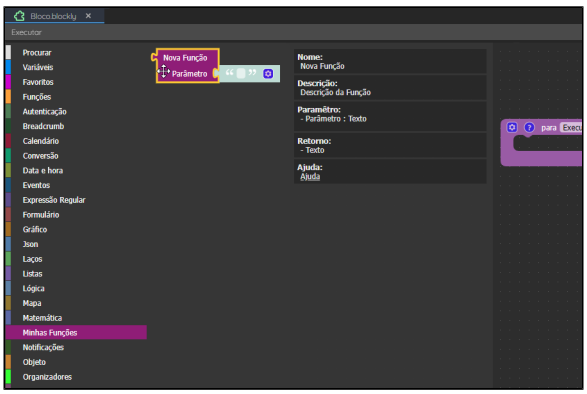

Figura 1.6 - Nova categoria e bloco de programação

Estrutura do código

Na figura abaixo vemos o código gerado a partir do arquivo JavaScript gerado no passo acima (Endereço: `src/main/webapp/js/novosBlocos/nomeArquivo.cronapi.js`).

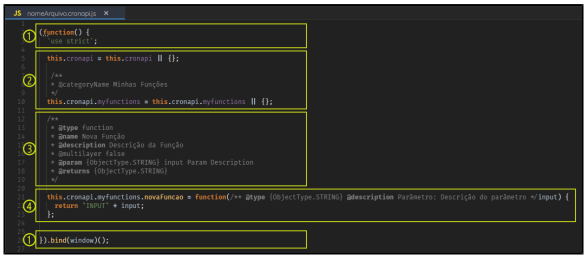

Figura 2 - Estrutura básica da função que gera o bloco de programação

1. **Função auto-executável**: todas novas funções de blocos devem estar dentro de uma função principal, ela é uma função anônima, auto executável e que faz referência ao escopo principal.
2. **Categoria**: o objeto `cronapi` comporta todas as funções JavaScript referente aos blocos clientes em suas respectivas categorias. Nesse momento é verificado se existe o objeto no escopo principal e caso não exista, é criado um vazio.
    - A anotação `@categoryName` informa o nome da categoria em que serão exibidos os novos blocos no editor Blockly.
    - É criado o objeto `myfunctions` dentro do `cronapi`, ele irá representar a nova categoria (**Minhas Funções**).
3. **Anotações da função**: acima da função são incluídas diversas anotações, cada anotação possui um objetivo próprio, descreveremos cada um mais a frente.
4. **Função JavaScript**: Além da função em si, no local dos parâmetros da função são incluídos mais algumas anotações, detalharemos mais a frente.

A cor da categoria e dos blocos contidos nela vão variar de acordo com o nome definido na anotação @category ou `@categoryName`. O Cronapp obtém o nome e gera uma chave hash para definir uma cor hexadecial. O objetivo é impedir que duas categorias possuam a mesma cor.

Assim, um categoria com o nome "Blocos novos" ficará com a cor #2e6f28, já o nome "Minhas Funções" receberá a cor #8f1d77.

Anotação

As anotações são palavras chaves que ficam comentadas no código JavaScript e são precedidas do caractere arroba "@". Elas têm o objetivo de renderizar um bloco de programação cliente que faz referência a uma função JavaScript (*high-code*), definindo características como nome amigável, descrição, formato do bloco e outros atributos.

Anotação: Categoria das funções

As anotações abaixo devem ficar acima da linha responsável por criar o objeto que irá conter as funções (blocos) relacionadas àquela categoria (destaque 2 da figura 2). Por sua vez, todas as funções (blocos) contidas nesse objeto estarão listadas nessa categoria no Editor de Blockly.

@Category

Representa o nome amigável que será exibido na categoria do Editor de bloco de programação. Essa anotação permite internacionalização.

Também é possível utilizar a anotação "`@CategoryName`".

Exemplo:

- Sem internacionalização:
  `@category Conversões métricas`
- Com internacionalização:
  `@category {{conversaoMetric}}`

@categoryTags

Essa anotação não possui ação atualmente, foi criada para ser utilizada no futuro, quando existirem uma quantidade relevante de categorias. Funcionará de forma semelhante ao `@nameTags`, permitindo adicionar uma lista de nomes para referenciar a categoria e ser filtrado no campo de busca das categorias. Utilize o caractere *pipeline* "|" para adicionar uma lista com diversos valores.

Exemplo:

```
@categoryTags device | dispositivo | info
```

Anotação: Função

As anotações da função devem ficar acima da linha da função (bloco) (destaque 3 da figura 2) e são responsáveis por definir algumas características do bloco de programação.

@deprecated

Indica que o bloco será depreciado, ou seja, ele será mantido para quem já está utilizando ele, mas não aparecerá para novos projetos. Só adicionar esse campo quando for depreciar um bloco, senão não adicionar.

Exemplo:

```
@deprecated true
```

@type

Define se a função irá representar um bloco de programação.

Valores:

- `function`: a função ficará exposta no editor de blockly, representada por um bloco de programação.
- `internal`: a função não será exposta no editor de blockly, será de uso interno, apenas por outras funções JavaScript.

Exemplo:

```
@type function
```

@name

Representa o nome amigável que será exibido no bloco de programação dentro do Editor de bloco de programação. Essa anotação permite internacionalização.

Exemplo:

- Sem internacionalização:
  `@name Minha Função`
- Com internacionalização:
  `@name {{minhaFunc}}`

@nameTags

Permite adicionar palavras chaves para referenciar o bloco no campo de busca das categorias. Utilize o caractere *pipeline* "|" para adicionar uma lista com diversos valores.

Exemplo:

```
@nameTags device | dispositivo | info
```

@description

Descrição sobre a funcionalidade do componente, será exibido na aba lateral da lista de blocos ou ao colocar o cursor do mouse em cima do bloco. Permite internacionalização.

Exemplo:

- Sem internacionalização:
  `@description Retorna o valor "INPUT" mais o valor do parâmetro.`
- Com internacionalização:
  `@description {{minhaFuncDescricao}}`

@multilayer

Define se o bloco estará disponível nos lados clientes e servidor (`true`) ou apenas do lado do cliente (`false`).

Exemplo:

```
@multilayer true
```

@platform

Permite que o bloco seja exibido nos dois projetos clientes no editor de blockly, ou apenas em um dos projetos. Caso deseje que o bloco seja exibido tanto no cliente web quanto no mobile, não adicione essa anotação.

Valores:

- `W`: será exibido apenas no editor de bloclky web.
- `M`: será exibido apenas no editor de bloclky mobile.

Exemplo:

```
@platform M
```

@arbitraryParams

Adiciona um ícone de engrenagem no bloco de programação (destaque 1 da figura 2.1) e permite abrir uma pequena janela onde é possível adicionar mais parâmetros ao bloco. Os valores dos novos parâmetros podem ser obtidos através do objeto `arguments` contido em todas as funções JavaScript. T anto o título quanto a descrição dos novos parâmetros adicionados, serão iguais ao último parâmetro padrão do bloco.

Valores: Essa anotação só aceita o valor `true`.

Exemplo:

```
@arbitraryParams true
```

No exemplo abaixo, o segundo parâmetro da função (`input2`) possui o nome "Demais Parâmetros", após arrastar mais 2 parâmetros ao container, esses novos parâmetros também receberão o nome de "Demais parâmetros".

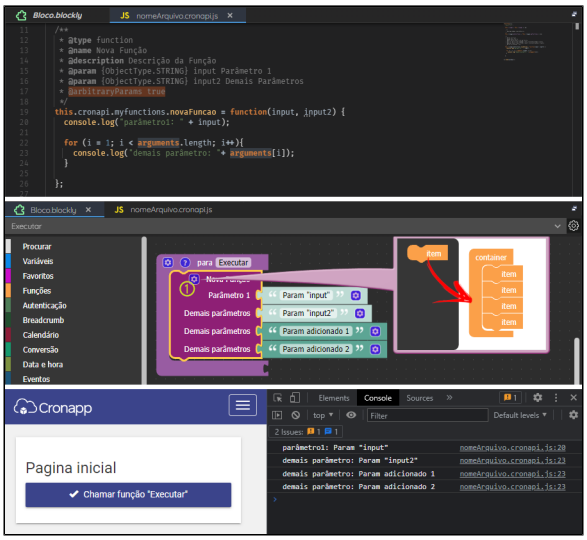

Figura 2.1 - Exemplo de uso da anotação @arbitraryParams

@wizard

As anotações wizards são utilizadas para gerar algum comportamento diferente ao bloco de programação. A criação das anotações `@wizard` é feita exclusivamente pela equipe de desenvolvimento do Cronapp e muitas vezes são utilizadas apenas para um bloco específico.

Dessa forma, não recomendamos o uso das anotações `@wizard` em suas funções personalizadas, pois ela poderá mudar totalmente o comportamento do bloco representado pela função.

Exemplo:

```
@wizard field_from_screen
```

@returns

Inclui e define o tipo de retorno do bloco. Ao adicionar essa anotação, o bloco receberá um encaixe lateral que permite ser encaixado nos parâmetros de outros blocos (destaque 1 da figura 2.2), caso não inclua essa anotação, o bloco será sem retorno (destaque 2).

Mesmo não adicionando a anotação `@returns`, algumas anotações forçam a criação do encaixe lateral (retorno), como e o caso do `@wizard`.

Valores:

- `{ObjectType.VOID}`: sem retorno.
- `{ObjectType.STRING}`: texto.
- `{ObjectType.BOOLEAN}`: boleano (true/false).
- `{ObjectType.LIST}`: lista.
- `{ObjectType.OBJECT}`: objeto.
- `{ObjectType.DATETIME}`: formato data e hora.
- `{ObjectType.LONG}`: inteiro longo.
- `{ObjectType.MAP}`: mapa (chave/valor)

Exemplo:

```
@returns {ObjectType.STRING}
```

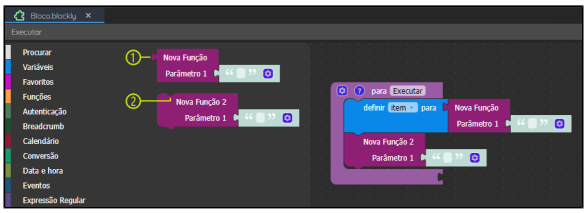

Figura 2.2 - Exemplo de blocos com e sem anotação @returns

@param

Essa anotação não é obrigatória e permite informar o tipo do parâmetro, apresentar um nome amigável e incluir uma descrição que será exibida na aba de descrição do bloco. Para definir outras características do parâmetro, será necessário configurar diretamente na anotação do parâmetros da função. A ordem das anotações deve ser idêntica a ordem dos parâmetros da função, ou seja, a primeira anotação `@param` deve referenciar o primeiro parâmetro da função e assim por diante. Permite internacionalização no nome low-code e descrição.

Ao definir um tipo ao parâmetro na anotação, o parâmetro do bloco poderá exibir um comportamento diferente. Por exemplo, ao definir um parâmetro como inteiro "`{ObjectType.LONG}`", o parâmetro do bloco exibirá uma caixa que só aceita números inteiros.

Sintaxe:

```
@param <tipo> <nome high-code> <nome low-code>:<descrição>
```

- **<tipo>**: descreve o tipo esperado pelo parâmetro, podendo modificar o parâmetro ou o bloco:
  - `{ObjectType.STRING}`: texto, exibe uma caixa de texto junto ao parâmetro, mas não impossibilita que outros blocos possam ser encaixados no local.
  - `{ObjectType.BOOLEAN}`: boleano (true/false).
  - `{ObjectType.LIST}`: lista.
  - `{ObjectType.OBJECT}`: objeto.
  - `{ObjectType.DATETIME}`: formato data e hora.
  - `{ObjectType.LONG}`: inteiro longo, exibe uma caixa que só aceita números junto ao parâmetro, mas não impossibilita que outros blocos possam ser encaixados no local.
  - `{ObjectType.MAP}`: mapa (chave/valor).
  - `{ObjectType.STATEMENT}`: inclui uma área de entrada de comando onde é possível adicionar e executar blocos sem retorno.
  - `{ObjectType.STATEMENTSENDER}`: inclui uma área de entrada de comando onde é possível executar uma função callback do JavaScript e utilizar seu retorno como parâmetro dos blocos encaixados na área de entrada de comando.
- **<nome high-code>**: nome do parâmetro da função.
- **<nome low-code>**: nome que será exibido no bloco de programação.
- <**descrição**>: descrição do parâmetro que será exibida na aba lateral do bloco

Exemplo:

- Sem internacionalização:
  `@param {ObjectType.STRING} input Entrada de texto: Informe um texto.`
- Com internacionalização:
  `@param {ObjectType.STRING} input {{entradaTexto}}`

@displayInline

Define como os parâmetros serão exibidos no bloco de programação. Não informando a anotação, o valor padrão será sempre `false`.

Valores:

- `true`: os parâmetros do bloco serão internos, esticando o bloco horizontalmente (destaque 1 da figura 2.3).
- `false`: (padrão) os parâmetros do bloco serão externos, esticando o bloco verticalmente (destaque 2 da figura 2.3).

Exemplo:

```
@displayInline true
```

Figura 2.3 - O primeiro bloco possui a anotação `@displayInline` e o segundo não

Anotação: Parâmetros

As anotações dos parâmetros devem ficar na mesma linha e antes de cada parâmetro da função que define o bloco dentro de um comentário do tipo bloco de código (destaque 4 da figura 2):

```
function(/** @anotações do param1 */ param1, /** @annotations  do param2
*/ param2){ ... }
```

Obrigatoriamente não deve conter quebra de linha nos parâmetros e em suas anotações na chamada da função, dessa forma, é esperado que funções JavaScript com muitos parâmetros fiquem com longas linhas.

@type

Informa qual o tipo de dados que o parâmetro espera receber. Essa anotação não é obrigatória nem impedirá que outros tipos sejam passados como parâmetro. Porém, ao definir um tipo do parâmetro na anotação, o parâmetro do bloco poderá exibir um comportamento diferente. Por exemplo, ao definir um parâmetro como inteiro "`{ObjectType.LONG}`", o parâmetro do bloco exibirá uma caixa que só aceita números inteiros.

Valores:

- `{ObjectType.STRING}`: texto, exibe uma caixa de texto junto ao parâmetro, mas não impossibilita que outros blocos possam ser encaixados no local.
- `{ObjectType.BOOLEAN}`: boleano (true/false).
- `{ObjectType.LIST}`: lista.
- `{ObjectType.OBJECT}`: objeto.
- `{ObjectType.DATETIME}`: formato data e hora.
- `{ObjectType.LONG}`: inteiro longo, exibe uma caixa que só aceita números junto ao parâmetro, mas não impossibilita que outros blocos possam ser encaixados no local.
- `{ObjectType.MAP}`: mapa (chave/valor).
- `{ObjectType.STATEMENT}`: inclui uma área de entrada de comando onde é possível adicionar e executar blocos sem retorno.
- `{ObjectType.STATEMENTSENDER}`: inclui uma área de entrada de comando onde é possível executar uma função *callback* do JavaScript e utilizar seu retorno como parâmetro dos blocos encaixados na área de entrada de comando (destaque 1 da figura 2.4).

Exemplo:

```
@type {ObjectType.BOOLEAN}
```

No exemplo abaixo, após consultar o nome da cidade a partir do parâmetro **CEP** (parâmetro tipo String), o bloco retorna o valor na variável "item", permitindo utilizar o retorno junto a outros blocos no parâmetro de entrada de comando **Sucesso** (parâmetro tipo `STATEMENTSENDER`).

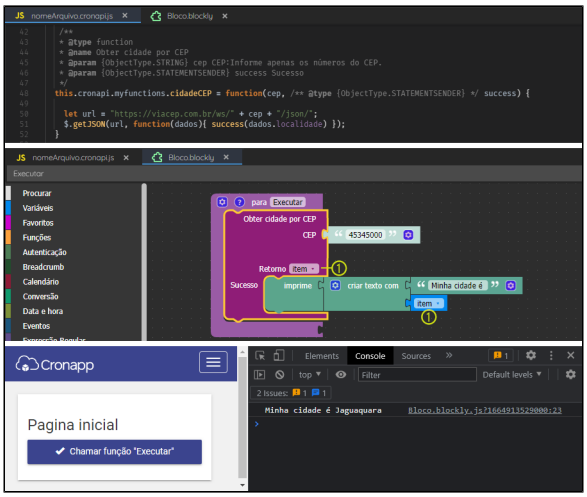

Figura 2.4 - Exemplo de uso do tipo `STATEMENTSENDER` em um parâmetro

@description

Adiciona uma descrição ao parâmetro da função e será útil apenas ao gerar documentações com o uso do JSDoc. Para incluir uma descrição que será exibida na aba lateral de descrição do bloco, utilize a anotação da função `@param`.

Exemplo:

- Sem internacionalização:
  `@description Entrada de texto`
- Com internacionalização:
  `@description {{entradaTexto}}`

@defaultValue

Define um valor padrão para o parâmetro, esse valor será visível para o usuário ao arrastar o bloco da sua categoria. Permite internacionalização.

Exemplo:

- Sem internacionalização:
  `@defaultValue Informe o nome da cidade`
- Com internacionalização:
  `@defaultValue {{BuscarCEPparamCidadeDescricao}}`

@blockType

As anotações `blockType` são utilizadas para anexar algum bloco, com retorno específico, nos parâmetros do bloco de programação. A criação das anotações `@blockType` é feita exclusivamente pela equipe de desenvolvimento do Cronapp e só recomendamos o uso das opções apresentadas aqui.

Com exceção do `@blockType util_dropdown`, que é personalizável, todos os demais existem separadamente na lista de blocos de programação e podem ser utilizamos como um facilitador para quem irá utilizar.

Entre a anotação @blockType e o valor, só pode haver um caractere de espaço em branco " ".

Valores:

- **`util_dropdown`**: exibe uma caixa de seleção com valores definidos nas anotações `@keys`, `@values` e `@defaultValue`.
- **`util_report_list`**: exibe uma caixa de seleção com os relatórios contidos no projeto, retornando o endereço do relatório selecionado.

- **`util_dashboard_list`**: exibe uma caixa de seleção com os dashboards contidos no projeto, retornando o endereço do dashboard selecionado.
- **`field_from_screen`**: exibe o bloco form_fieldscreen_callreturn que lista todos os elementos que possuem a propriedade ng-model na tela. É necessário referenciar um formulário ao editor de bloco de programação, veja mais detalhes no tópico "Propriedades do Blockly" em Bloco de programação.
- **`datasource_from_screen`**: exibe o bloco Fonte de Dados que lista todos os *datasources* na tela. É necessário referenciar um formulário ao editor de bloco de programação, veja mais detalhes no tópico "Propriedades do Blockly" em Bloco de programação.
- **`ids_from_screen`**: exibe o bloco Obter identificador do componente que lista todos os elementos HTML que possuem o atributo id na tela. É necessário referenciar um formulário ao editor de bloco de programação, veja mais detalhes no tópico "Propriedades do Blockly" em Bloco de programação.
- **`variables_get`**: exibe o bloco Obter variável que lista todas as variáveis utilizadas na função do Bloco de programação.

Exemplo:

```
@blockType ids_from_screen
```

No exemplo abaixo, o bloco de programação **Nova Função 3** possui 2 parâmetros configurados com anotações `blockTypes` diferentes: **Id**, possui um bloco que retorna todos os identificadores dos elementos da página referenciada nas configurações do editor de bloco de programação e **Relatório**, que listará todos os relatórios contidos no projeto. O bloco **Nova Função 3** apenas imprime o conteúdo retornado pelos blocos anexados aos parâmetros.

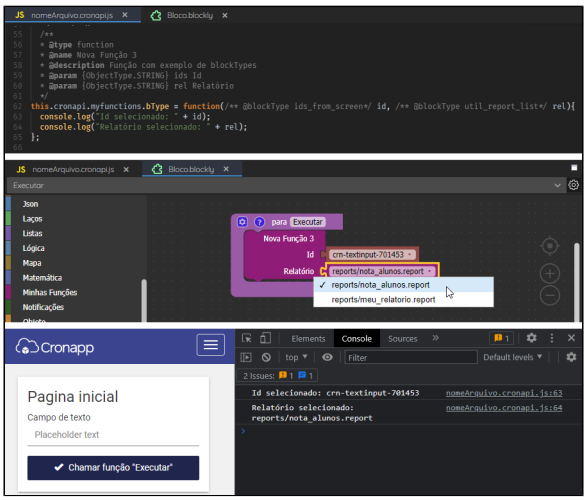

Figura 2.5 - Exemplo de uso da anotação `blockTypes` em um parâmetro

@keys

Essa anotação deve ser utilizada em conjunto com a anotação `@blockType util_dropdown` e permite listar diversos valores na caixa de seleção anexada ao parâmetro, a caixa de seleção retornará o valor selecionado pelo usuário. Utilize o caractere *pipeline* "|" para adicionar uma lista com diversos valores. Não é possível internacionalizar o conteúdo dessa lista, para isso, utilize a anotação `@values`.

Exemplo:

```
blockType util_dropdown @keys smartphone|tablet|monitor|tv
```

@values

Essa anotação deve ser utilizada em conjunto com as anotações `@blockType util_dropdown` e `@keys` e permite exibir nomes amigáveis na lista da caixa de seleção anexada ao parâmetro. Após selecionar um item com o nome amigável na caixa de seleção, o valor passado para o parâmetro da função será o valor equivalente na anotação `@keys`. Utilize o caractere *pipeline* "|" para adicionar uma lista com diversos valores.

A quantidade de elementos das anotações `@keys` e `@values` devem ser iguais, e o primeiro elemento da anotação `@keys` deve referenciar o primeiro elemento da anotação @values e assim por diante.

Para apresentar opções que possuam caracteres especiais ou espaços em branco " ", será necessário internacionalizar o conteúdo em `@values`.

Exemplo:

```
/**
* @type function
* @name Nova Função 5
* @description Função com exemplo das anotações @blockType util_dropdown,
@keys e @values
* @param {ObjectType.STRING} opcao Equipamento
*/
this.cronapi.myfunctions.dropDown = function(/** @blockType util_dropdown
@keys smartphone|tablet|monitor|tv @values {{dispPequeno}}|{{dispMedio}}
|Desktops|{{dispGrande}} */ opcao){
  console.log("Equipamento selecionado: " + opcao);

};
```

No exemplo acima, apenas o `@values` "Desktops" (referenciando a chave "tablet") não possui internacionalização. A execução do bloco pode ser visualizada na figura abaixo.

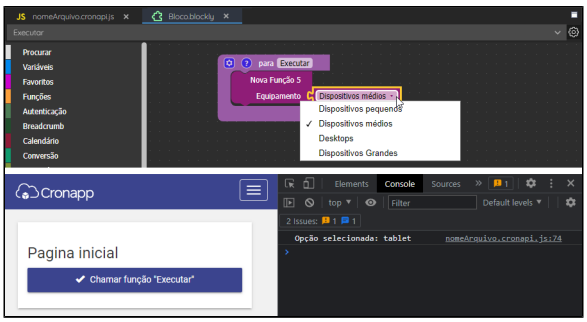

Figura 2.5 - Exemplo de uso das anotações `@values` e `@keys` em um parâmetro

Internacionalização

Todas as anotações utilizadas para exibir algum valor no bloco de programação, como o nome do bloco e seus parâmetros, descrições e lista de opções, permitem a internacionalização. O idioma exibido no bloco será o mesmo definido pelo usuário no Cronapp Studio, essa configuração é feita a partir do menu do sistema **Espaço de Trabalho > Idiomas**, podendo selecionar as opções "Português" ou "Inglês".

Diferentemente das aplicações desenvolvidas no Cronapp que suportam todos os idiomas, o conteúdo desenvolvido para ser utilizado dentro do Cronapp, como bloco de programação ou componente visual, só poderão ser internacionalizados nos idiomas português e inglês.

Para conseguir internacionalizar os blocos de programação, uma pasta com o nome "`i18n`" (destaque 2 da figura 3) deve ser criada no mesmo diretório (destaque 1) onde está o arquivo JavaScript que vão gerar os blocos de programação (destaque 3). A pasta "`i18n`" deve conter os arquivos de internacionalização: "locale_en_us.json" e "locale_pt_br.json". Ao abrir os arquivos de internacionalização JSON, a ferramenta Chave de internacionalização é exibida, permitindo adicionar as novas chaves e traduzi-las automaticamente.

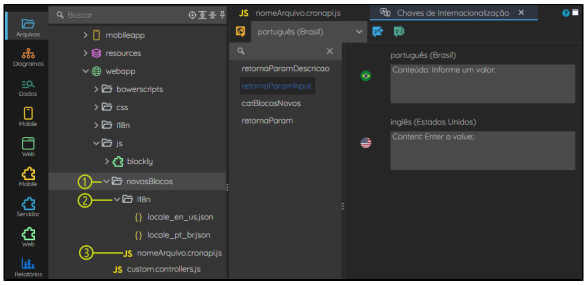

Figura 3 - Local do diretório i18n em relação ao arquivo JavaScript e ferramenta que facilita a inter nacionalização em vários idiomas

A chave de internacionalização no código JavaScript deve estar ao lado das anotações que aceitam esse recurso e dentro de 2 caracteres chave "{", exemplo: "`{{valorDaChave}}`". Ao abrir o Editor de Bloco de programação, o Cronapp irá verificar a chave informada, e retornar o valor informado nos arquivos de internacionalização.

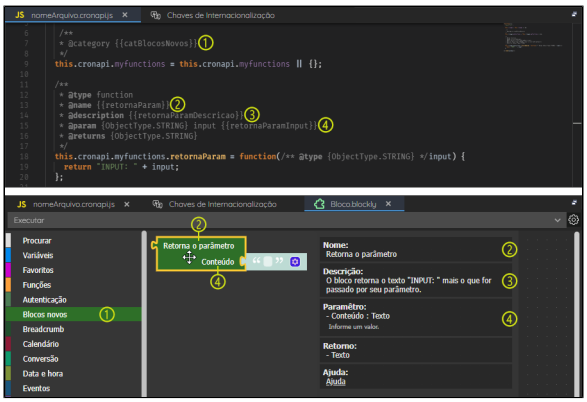

Figura 3.1 - Anotações com as chaves de internacionalização e onde interfere no editor de Blockly

1. Categoria dos blocos. Como informado no tópico Estrutura do código, o nome da categoria definirá a cor dos blocos da categoria.
2. Nome do bloco.
3. Descrição do bloco.
4. Nome e descrição do parâmetro.