Blockly Cerebrum Documentation: Table of Contents

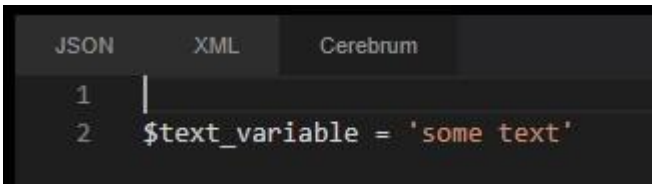
Basic Usage

Drag blocks from the categories on the left hand side of the playground into the playground. It should be visually obvious what kinds of blocks are eligible connections to each other. If it isn't we should review Blockly's basic playground and learn about how to use Blockly.

- The following is an example of assigning a variable to a text value. Note that we can use other data types as the variable's definition.
- To create a variable select the Variables category, click "Create Variable" and name the variable. In the same Variables category, we can now drag a variable set block into the workspace.

- We can connect these two blocks to set text_variable to "some text"

As we work in the Blockly workspace, with Cerebrum selected in the generators section, we will see Cerebrum code generated based on the blocks in our workspace. Below is the corresponding generator display when we connect the blocks shown above.

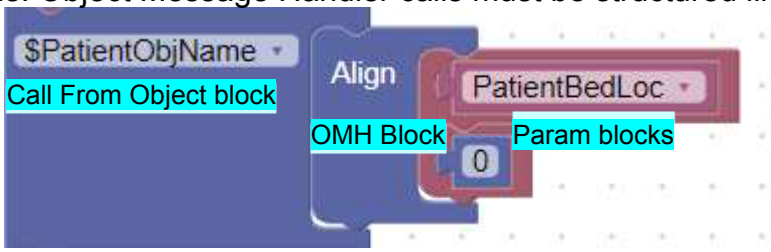


This is a single example of the general flow of using the editor and what we can expect from it as output.

The editor contains a toolbox on the left where we can drag and drop new blocks into the workspace. The toolbox is broken apart into several categories:

- **Input**
  Contains blocks for simple input types: Number, String, List
- **Expression**
  Contains blocks for us to create logical or arithmetic expressions
- **Logic**
  Contains the following blocks: if, if else, logical expression, compound logical expression, not, boolean
- **Comment**
  Contains blocks for adding comments to code
- **Variables**
  Contains blocks that allow us to create new variables or edit the value of an existing variable

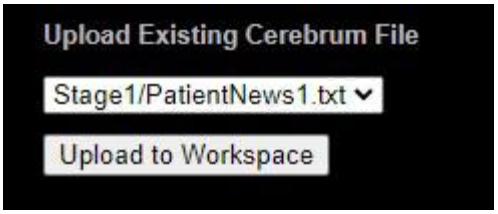
- **Do**
  Contains several different versions of a "Do" command as well as GoTo commands. GoTo and Do blocks both have a Return variant as well as a Does Not Return variant. This section of the toolbox also contains a Do block that allows us to reference a function path directly if we have one.
- **Functions**
  Contains all existing functions defined in the workspace. This section of the toolbox also allows us to create and define new functions.
- **Call From Object**
  The block in the Call From Object section is necessary for Object Message Handler calls. Object Message Handler calls must be structured like so:



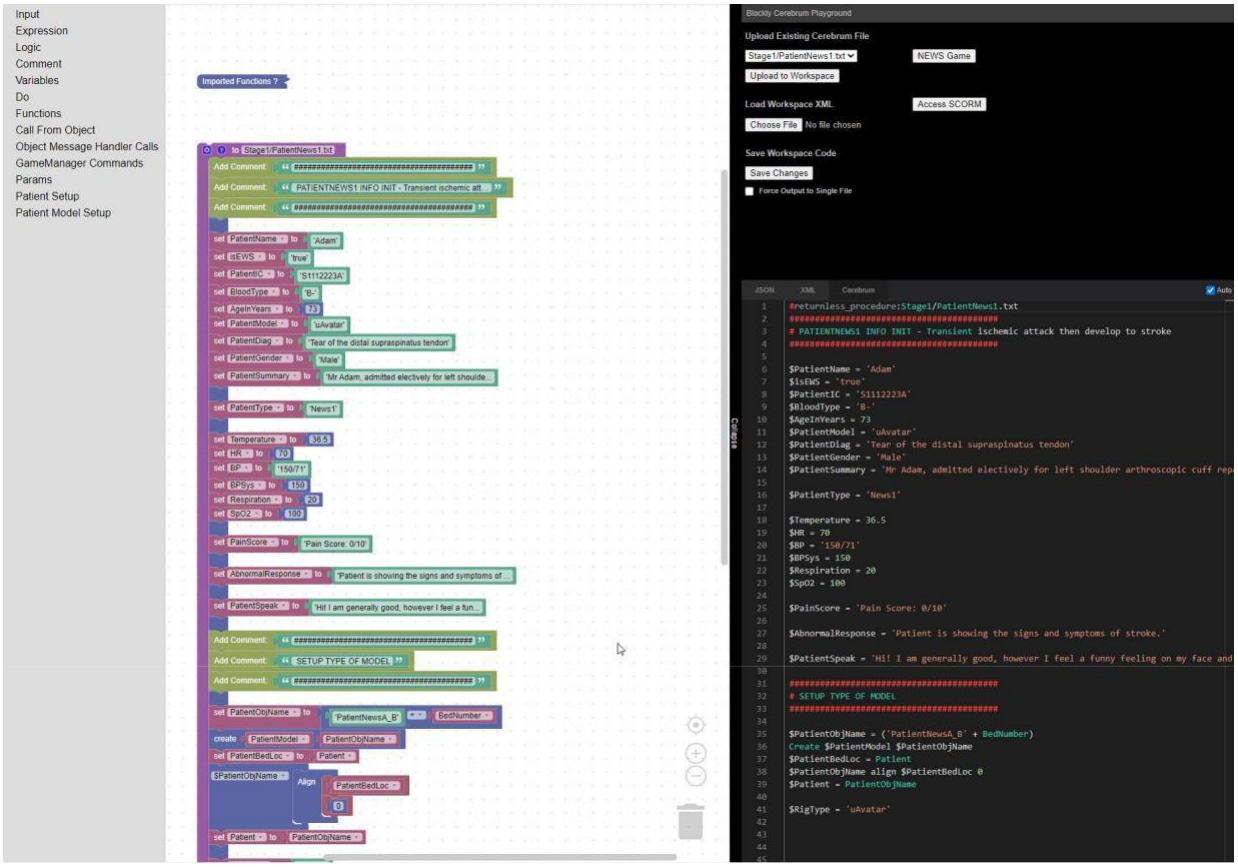
- **Object Message Handler Calls**
  Contains all Object Message Handler calls. These must be attached to a "Call from Object" block as shown above. Some Object Message Handler blocks require param inputs which we can see in the example above. These param inputs can be found in the Params section of the toolbox.
- **GameManager Commands**
  Contains blocks for calling a GameManager Command. Some Object Message Handler blocks require param inputs which we can see in the example above. These param inputs can be found in the Params section of the toolbox.
- **Params**
  The Param block is necessary for some OMH calls and for some GameManager calls. Inside the Param block (as illustrated in the example above) we have to put an input block. In the case above, "PatientBedLoc" is a variable block that we use as input in the first Param block. Attached to this is a second param block that we attach a numeric input to (0 in the case above).
- **Patient Setup**
  Contains preset stack of blocks for patient setup. This large collection of blocks contains information about the patient like vitals, name, etc.
- **Patient Model Setup**
  Contains preset stack of blocks for the model setup of a patient.
  Creating a new patient requires both a Patient Setup preset block and this Patient Model Setup block.

Interacting With Existing Cerebrum Files

The Cerebrum Blockly Editor can be connected to an existing library of Cerebrum files. We can interact with these files and upload them to the workspace where we can make and save changes.



When we select a file from the dropdown and upload it to the workspace, we should see a large number of connected blocks that make up whatever file we've selected.



We can see that the file we've selected has been imported to the workspace. Its contents are represented in blocks in the workspace and the corresponding Cerebrum code that is generated is shown on the right.

We can make changes to the values of any of the blocks that have been generated or add new ones and connect them. Our changes will be visible in the generator section. Once we're done making changes to the workspace, we can use the Save Changes button to download the updated file and save it locally. If you are using SGAsia's implementation of the editor this button will save the changes you've made remotely to the AWS server where the files are stored instead of downloading the revised server.

The following video might prove helpful for reference: https://youtu.be/03TEnb0E9

Installing and Setting Up a New Editor Implementation

First we need to download the Cerebrum Editor. It can be downloaded from: https://github.com/adamburich/Blockly_Cerebrum as a zip file. Unzip it and note the location of the directory on your computer.

Now we can navigate to this directory and issue the following command in terminal:

```
PS D:\Documents\Blockly_Cerebrum> npm run start
```

This will start a node server which serves our Cerebrum Blockly implementation. By default the basic playground should open. The path we want to go to for the Cerebrum Editor is: "/tests/playgrounds/advanced_playground.html"

Note that for security purposes installations from the github link above we do not provide write access to SGAsia's servers which host the Cerebrum files we will interact with via the editor UI. We can still pull files and view them in the workspace but no changes we make will write and save to their servers.

Creating and Handling New Block Types

The core of this process is again the same as it is for native Blockly (with some caveats). If we are creating a new block type AND our block type is NOT an Object Message Handler call block or a Game Manager call block, we should follow Blockly's documentation for adding Blocks to the library and adding handling for the generator for those blocks.

Much of the block creation process can and should be done at: https://blockly-demo.appspot.com/static/demos/blockfactory/index.html

If we are creating an ObjectMessageHandler call block:

- Follow the instructions above and provided by Blockly's documentation. Add the definition for the block to Cerebrum/Blocks/ObjectMessageHandlerBlocks.js (https://developers.google.com/blockly/guides/create-custom-blocks/overview)
- Add the generator code to Cerebrum/Generator/BindObjectMessageHandlers.js
- Add the new Block type's name as a string to the block registry array in Cerebrum/Helper/Builders/OMH_Calls.js

If we are creating a GameManager call block:

- Follow the instructions above and provided by Blockly's documentation. Add the definition for the block to Cerebrum/Blocks/GameManagerCallBlocks.js (https://developers.google.com/blockly/guides/create-custom-blocks/overview)
- Add the generator code to Cerebrum/Generator/BindGameManagerCalls.js
- Add the new Block type's name as a string to the block registry array in Cerebrum/Helper/Builders/GameManager_RWORDS.js

In all other cases Blockly's default pattern of implementation (detailed here: https://developers.google.com/blockly/guides/create-custom-blocks/overview) should suffice for building new block types. We can put the definitions for new blocks in whichever of the existing files in Cerebrum/Blocks is most appropriate. If necessary we can create new files for new types of blocks.

- Each of these files contains a dictionary of block definitions that we need to export. We can import it in the CustomBlockLibrary.js file in Cerebrum/Blocks.
- If you make a new file to place blocks in you must complete this process.

Likewise, we must add the generator definition for these new blocks in Cerebrum/Generator files. We can use whichever of the existing files is most appropriate.

- Each of these files contains a single function, bindBlockLib (named differently as is appropriate for the contents). This function must then be exported and imported in Cerebrum/Generator/CerebrumGenerator.mjs
- We must call this bindBlockLib function using the generator object in CerebrumGenerator.mjs.
- If you make a new file to place generator code in you must complete this process.