Developer Documentation

[**1. Getting started with FlaxBlocks 2**](#_jmwapobp3bdj)

[1.1. Core idea 2](#_bvsgd3u42eu1)

[1.2. Requirements 2](#_da070uuhhyc5)

[1.3. Repository structure 2](#_ilc1hokzm8v5)

[**2. Main loop 3**](#_fzjxktoih73c)

[2.1. Initialization 3](#_lyt77cdxdno5)

[2.2. Block Creation WorkFlow 3](#_hl2yjwha08ia)

[2.3. npm run blockify 4](#_ps09mmdfdiw9)

[2.4. CheatBlocks 5](#_7xrrdzt1aoo8)

[2.5. workspaceInit 6](#_35xq786jf52p)

[2.6. Saving changes 7](#_z4qh95q5t0v2)

[2.7. Compare 7](#_bk2l6zwup45m)

[**3. Possible future 8**](#_vmqnoucmsq)

[3.1. Likely to change 8](#_jc38osf76rpt)

[3.2. Further ideas 8](#_s0edy4hgh3tb)

## 1. **Getting started with FlaxBlocks**

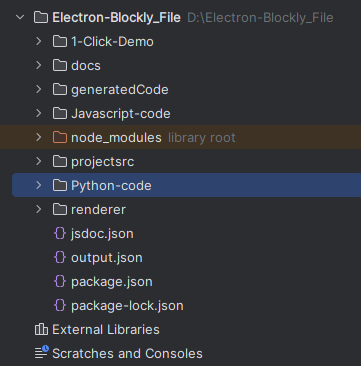
### Core idea

* + 1. FlaxBlocks is striving to reduce the entrance barrier from Jax/Flax, the key idea behind our application is to display code more understandably and intuitively. To accomplish this, we’re using [Blockly](https://developers.google.com/blockly/guides/create-custom-blocks/blockly-developer-tools?hl=de), Blockly lets anyone display code as Blocks - similar to Scratch-, thus giving the benefit of working with complex code more intuitively.

### Requirements

* + 1. Our project is run as a modern Client application, for this, we are using a node.js featuring Electron, we’ve decided to use Electron since it allows us to read/write files on our local devices. After pulling the provided repository, simply go to the main root and install all dependencies by typing   
       npm install in your terminal.   
       Finishing these steps, allows you to run the project with one of the given starting scripts e.g. npm run dev.   
         
       This requires that node.js is running on your device

### Repository structure

* + 1. node\_modules  
       Once downloaded the structure of the repository should look like this, (the node\_modules folder is created after running npm install, its advised to delete the file before pushing changes since its horrendously big)  
       
    2. Javascript-code  
       Here the user will find all relevant files handling javascript code, note that also the files created by installing Blockly are stored in a subfolder here. You will need to adjust these files if you want to create Blocks, the two most important files are *main.js* and *workspaceInit.js.*   
       Main.js handles the server side and workspaceInit the client side.
    3. Python-code  
       All relevant Python files are stored here, the two most important ones are *patternMatch.py* and *Watchdog.py.*
    4. generatedCode  
       Views and code which is generated are stored here as .py files, whenever the project is started this folder will be empty since we’ve deleted the content of it
    5. Projectsrc  
       Here two files can be found, projectsrc.py and projectskeleton.py.  
       The projectsrc.py will receive the full source code of a jax/flax script.  
       As the name implies, the projectskeleton.py will only be a skeleton of the previous file, hence only storing the contents which couldn’t be translatable.
    6. Renderer  
       Everything related to the styling and display of the project is found here, hence all files related to the index.html file.

## Main loop

### [Initialization](https://www.vocabulary.com/dictionary/initialization)

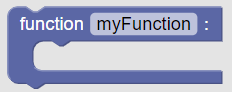
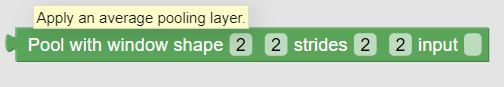
* + 1. The magic for creating the blocks happens between patternMatcher.py and workspaceInit.js, when initially a file is selected the content will be saved in the projectsrc.py, and then we start with our first script. After clicking on load-File, the client sends the ‘start-script’ event to the server. Thus the following process is started.

### Block Creation WorkFlow

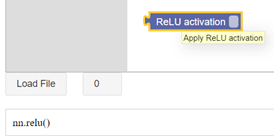
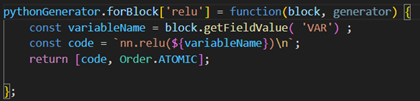
The process of custom block creation within the FlaxBlocks application involves multiple core files that work together to allow users to define and generate neural network code using Blockly. Here's a detailed description of the workflow and the core files involved:

2.2.1. Core Files and Their Roles

The **core files** of the FlaxBlocks system play an interconnected role in defining, configuring, and generating neural network components using custom blocks. The `blocks.js` file is central to defining the structure and shape of custom blocks. Each block's shape, inputs, outputs, and visual configuration, are defined here. For example, blocks like `relu`, `nn\_compact`, and `python\_function` are designed with specific inputs (e.g., text fields, colour inputs) and outputs (e.g., return values), which determine how these blocks interact with other blocks in the workspace. The shape of a block influences not only its appearance but also how it connects with other blocks, thus enabling seamless block-based programming.



|  | Once the blocks are defined in `blocks.js`, they are made available in the Blockly user interface through `toolbox.js`. This file configures which blocks are visible and categorizes them for easy access by users. Categories like "Layers," "Activation Functions," "Decorators" and others group blocks by their function, making it simpler for users to drag and drop them into their workspace. The `toolbox.js` file ensures that the blocks defined in `blocks.js` are organized and presented in a way that enhances usability.  Finally, `javascript.js` is responsible for turning these custom blocks into actual code. It maps each block's visual structure to the corresponding code using the `pythonGenerator.forBlock` functions. When a block is dropped into the workspace, `javascript.js` generates the corresponding Python or JAX/Flax code, such as a neural network layer or activation function. This file completes the process by ensuring that the blocks defined in `blocks.js` and configured in `toolbox.js` produce the desired code output, making the custom blocks functional components within the neural network framework. |
| --- | --- |



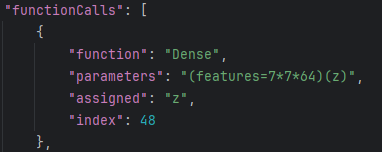
### npm run blockify

* + 1. After the file is loaded, we will start the patternMatch.py script, this script will look at the content of the projectsrc.py file.  
       Here it will use Tree-sitter to efficiently query the AST tree, this can be confusing at first, it's advised to look at the Tree-sitter playground to understand how it works.  
       The highlighted part is the node in the AST tree, that's also how the patternMatch works, we check in which scope we are, hence Class/Function and look for functions.  
       Here, whenever we encounter a function Call which is not *simple* - meaning, if the current row is not a simple assignment like x = 5-, it will check if we can translate this part of the function.   
       We do this by checking if the function identifier, in our example foo(), is present in the BlockFunction.py, as discussed, all Blockifiable functions are stored as an Enum here. Each scope which was translatable is deleted from the file and replaced with comments of the scope, these comments must have the pattern #Filename+

### CheatBlocks

* + 1. Bigger scopes often have function calls which reference functions declared in the same file, yet, this is not handled by the current application.   
       To avoid the scenario that most of the scope is translatable, yet one block isn’t hence the entire scope won’t be displayed as Blocks we’ve implemented cheatBlocks. After starting the project, they can decide how many blocks are allowed to be cheated. Cheated means blocks which simply store the entire LOC in one Block. By doing this, even though the Function is not translatable, the function will still be displayed as a Block.   
         
         
         
         
         
         
         
         
         
         
         
         
         
         
         
         
         
         
         
         
         
         
         
         
         
       

### workspaceInit

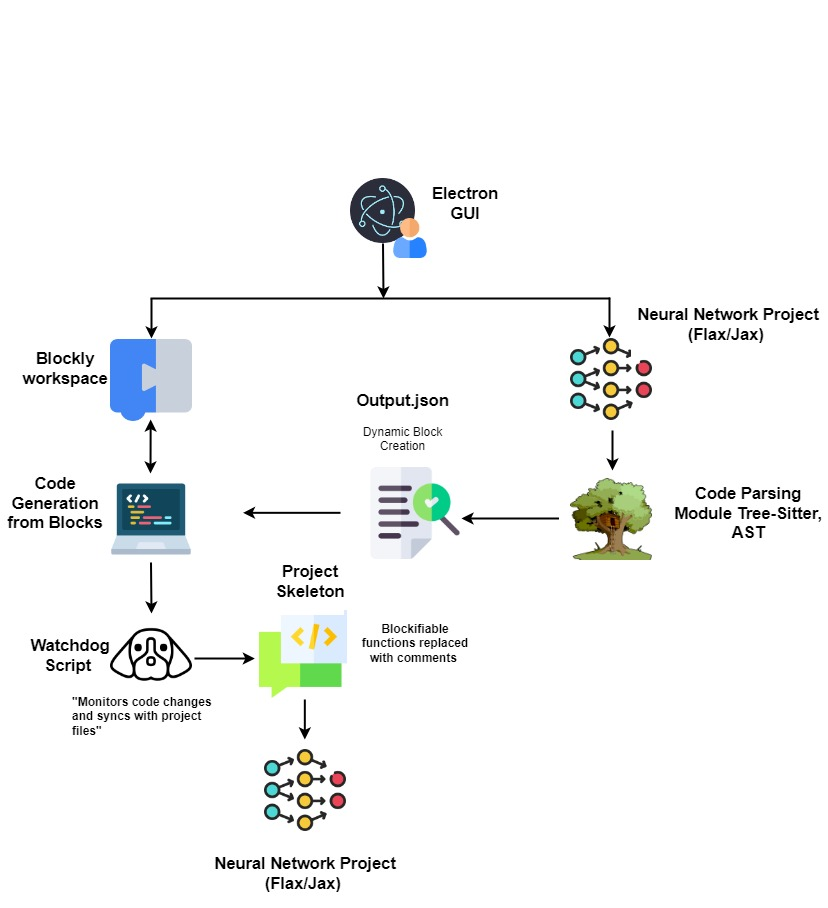
* + 1. Now, after we’ve successfully run npm run blockify, we will send another event, this time, we’ll tell the client can start creating blocks.   
       In our workspaceInit, we query the outputJson and check which type of block is needed for this JSON object.   
       In this example, we see the structure of a function calls, we use “function” to create the corresponding block and parameters to parse them into our Block.  
       If a functionCall has an assigned 

value, hence is not “ “, we create a variable Block and connect them.  
Since most people never worked with Blockly, its advised to take a look in the 1-Click-Demo folder, here some code snippets are provided that explain how to work with Blockly.   
After finishing this process, different views should be visible in the select view, here the user can change between different scopes (Classes/Functions) and change code directly in Blockly. Furthermore, for each scope, a file in “generatedCode” will be created which is empty at the start

### Saving changes

* + 1. After clicking “show changes”, the workspace init, will fire another event which saves all workspaces as code in the corresponding Files, our Watchdog.py is waiting for changes, whenever a file in the just named Directory is changed, it will insert the code in the corresponding Part of the projectskeleton, here the comments are important s.t the watchDog can insert the snippets at the correct position.

### Compare

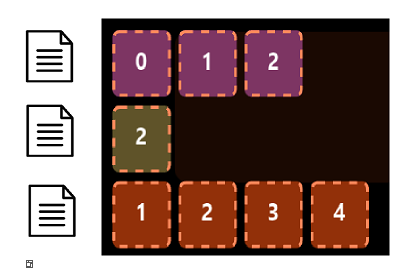
* + 1. Now the user can compare between the initially loaded file and the current one, if wanted, pressing on submit will insert the updated code in the initially chosen file.  
       *Note: Since we store the Path of the selected File at the start of the script, the file shouldn't be moved around, this will cause some unexpected error*

## Possible future

### Likely to change

* + 1. The most important thing which can be improved and changed in the future is the patchernMatch.py script, for example, currently, we don’t feature heavily wired function calls, hence not all functions which we may feature are created since the patternMatch.py does not recognize them.   
       Furthermore, one could advance the script in such a way that it's able to create scopes for if statements. Implementing this shouldn’t be that difficult, since the same logic is used for detecting *for e in range(arr)* types of loops.   
       Another key change which could improve the overall project is to try to run the patternMatch.py with the “context” of the file, hence checking whether a function which is called has just been defined in the file. Since by doing this, one could reference the function as a Block which leads to the other scope, thus creating a better user experience.

### Further ideas

* + 1. Scanning the entire directory instead of one file should be possible.  
       One workspace corresponds to one view, hence scope, these workspaces are stored in an array.  
        One could advance it in such a way, that we use an nested List, where we store all important files with their corresponding translatable scopes. We would suggest a nested List over an simple 2D array since the amount of scopes may vary between files.   
         
       This could then be represented as another select statement in the UI where the user can choose between the current File.  
         
       3.2.2. Currently, the project focuses on model definition using the Flax and JAX packages primarily utilized for building neural network components. However, there is potential to extend the project to include features like training workflows, data loading, and evaluation. Furthermore, the software could be expanded to support other neural network libraries such as PyTorch and TensorFlow, making it a versatile platform for visual neural network development across various frameworks.