Music Blocks 4 Project Builder Integration

Basic Details

Name: Husain Shahid Rao

Email: husainshahidrao@gmail.com

Github: https://github.com/husain3012

Language: English, Hindi

Location: India, (IST, GMT+5:30)

https://www.husainshahidrao.com/

Open Source Contributions

- Sugarlab - MusicBlocks
    - https://github.com/sugarlabs/musicblocks/pull/3203
    - https://github.com/sugarlabs/musicblocks-v4-builder-framework/pull/5
- Other Contributions and work:
    - https://github.com/husain3012/automatasim
    - https://github.com/Greenstand/treetracker-admin-client/pull/246

Project Details

Music Blocks 4 Project Builder Integration

Building project builder as a wrapper component from the prototype, and adding functionality to access musicblocks-v4-lib APIs.

It would provide sugar labs with an all-new musisblock-v4, which will be much smoother and more robust than its predecessor.

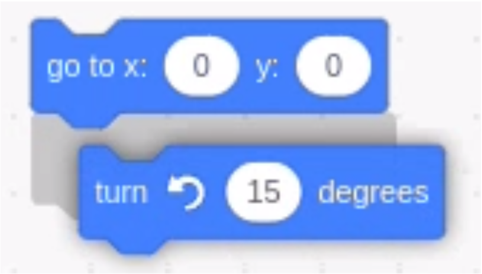
General Objectives:

- Refactor the prototype code
- Integrate it in `musicblocks-v4`
- Create a wrapper component *Project Builder* (*builder*) in `musicblocks-v4`
- Add utilities to the wrapper component so that the *Project Builder* component can communicate with the *Specification* and *Syntax Tree* APIs of the **Programming Framework**
- Create a *Palette* (*palette*) component

Proposed Solution

Refactoring the prototype

- Improving design of blocks, take inspiration from MIT's Scratch

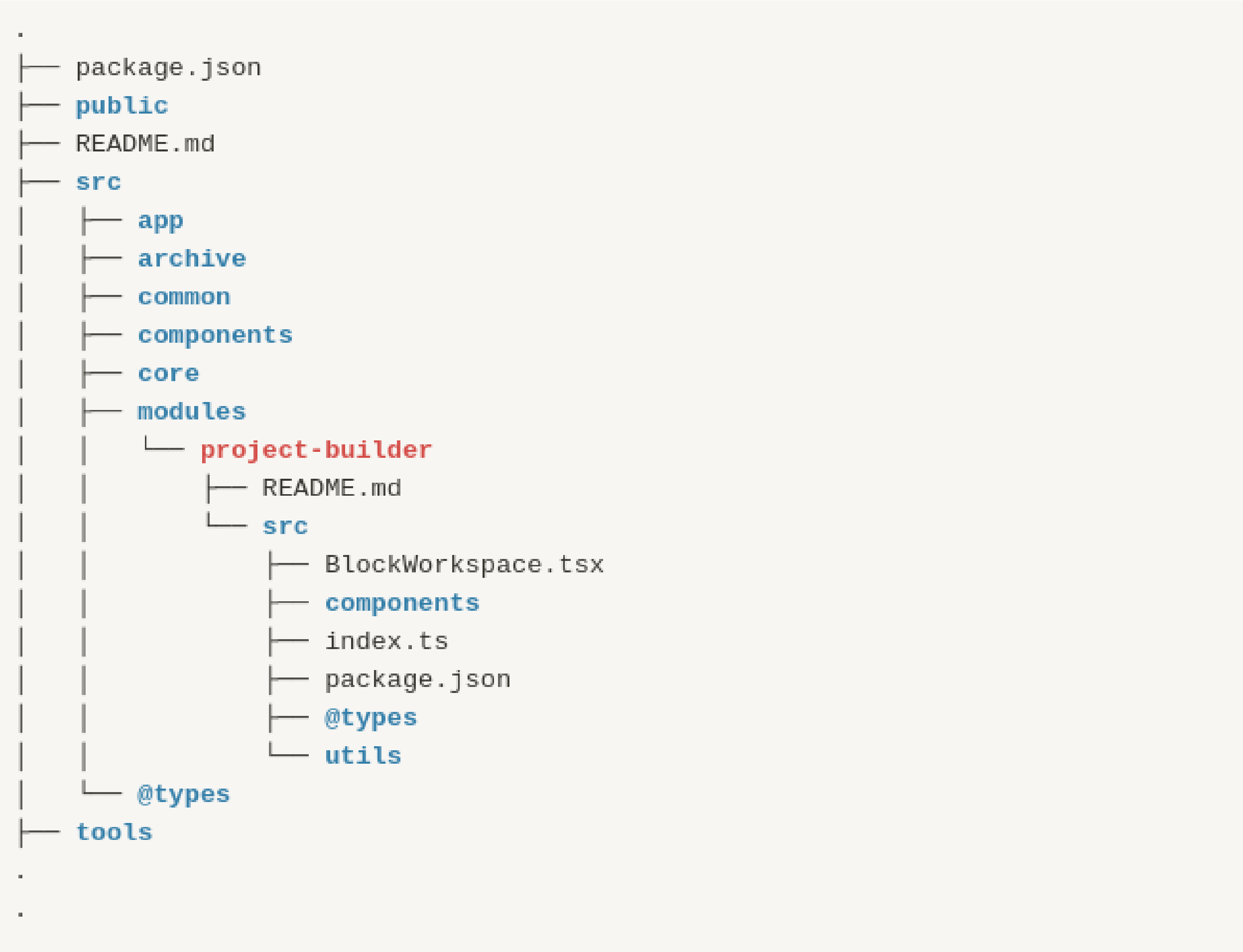


  - Material design and colors for various blocks
  - Highlighting drop-zones
  - Canvas with scrollable view and zoom in/out controls, etc.

- Making project suitable for to be used as a sub-module

  Remove boilerplate code for the standalone project builder application, and add necessary exports. These exports will include the `wrapper component` for the builder and other functions to tap into the state management of the builder.

- Proposed directory structure:



```
.
├── package.json
├── public
├── README.md
├── src
│   ├── app
│   ├── archive
│   ├── common
│   ├── components
│   ├── core
│   ├── modules
│   │   └── project-builder
│   │       ├── README.md
│   │       └── src
│   │           ├── BlockWorkspace.tsx
│   │           ├── components
│   │           ├── index.ts
│   │           ├── package.json
│   │           ├── @types
│   │           └── utils
│   └── @types
├── tools
.
.
```

Creating a wrapper component

- Handling app (project-builder) state via **redux** or **context api**:
    - Currently blocks are being stored in a redux store in a simple object `workspace` of the type `{ [id: string]: Block }`.

  Two approaches:

    - Use **context-api** instead
        - We can remove redux from the builder completely and can use react context api instead with the help of `useContext` hook.

    - This will prevent any kind of conflicts with any future use of redux in the main `musicblocks4` app.

  - Use **redux**
    - We can go with the current implementation of redux.
    - This will require us to configure a redux store in the main musicblocks app, and will result in tight coupling of the project-builder with the main app.

- We can export the `BlockWorkspace` as a wrapper component, and along with it, functions to get the current workspace state will also be exported.

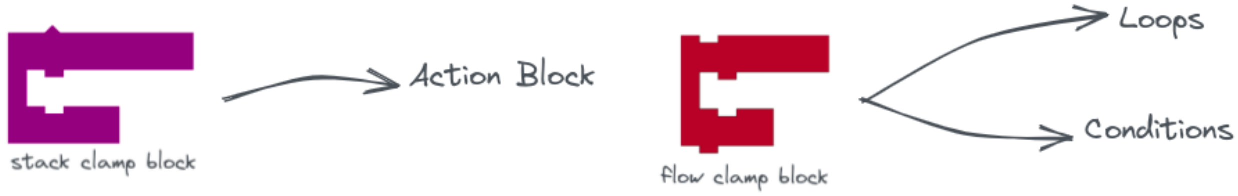
- So the imports will look something like:

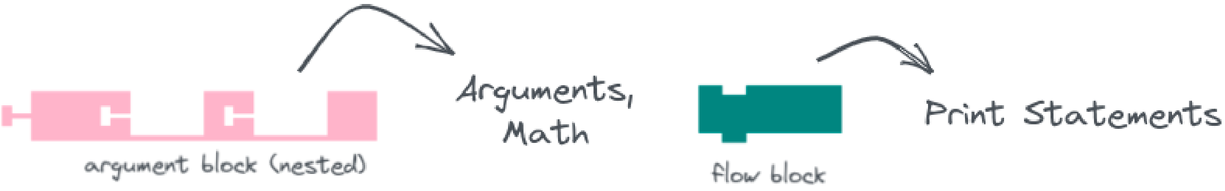
```
import ProjectBuilder from "@sugarlabs/musicblocks-v4-builder-framework"
import {getBuilderState} from "@sugarlabs/musicblocks-v4-builder-framework"
```

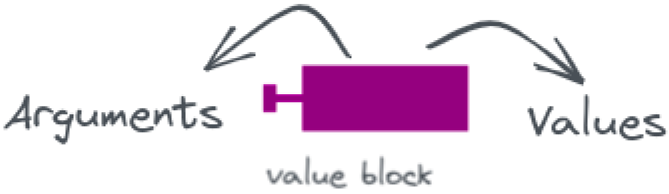
- `getBuilderState` will return the current blocks on the canvas.

Integration of Project builder with Programming Framework

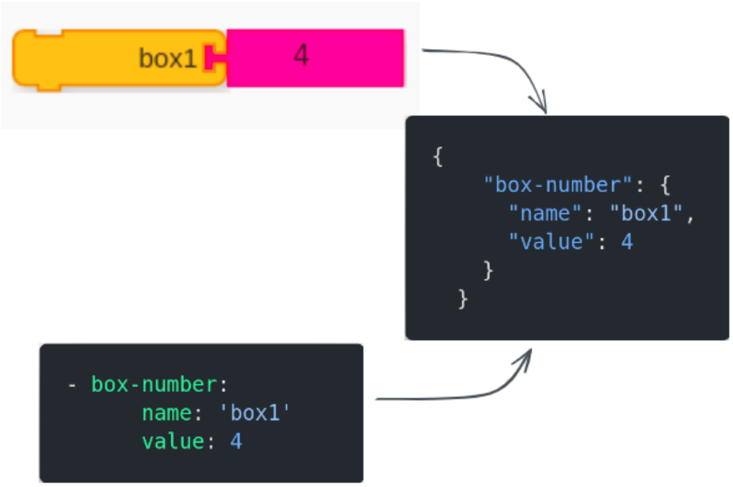
Brick Tree to Syntax Tree:







- For each type of **core element** in programming framework, we need a almost one-to-one mapping to the block elements on the canvas.
- Currently, in `musicblocksv4` YAML code is given in editor, which is fed into the `buildProgram()` function, where, it is converted into JSON and `programming-framework` APIs are used to build the programming.
- On creating mapping between blocks we can get the list of current instructions directly by tapping into the `project-builder` state.
- **Examples:**

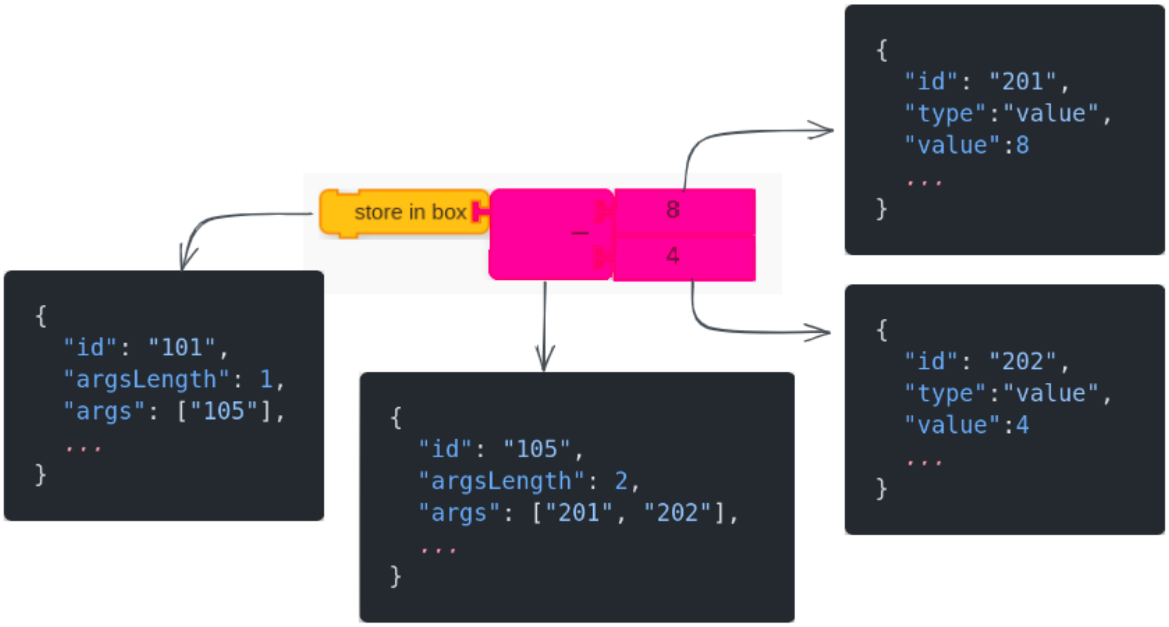


```
{
    "box-number": {
      "name": "box1",
      "value": 4
    }
  }
```

```
- box-number:
      name: 'box1'
      value: 4
```

“box” block has info in JSON about it’s name and value

- `value` can be further nested into mathematical operations.
    - “value” block can directly return values
    - For nested operations, `args` can be **resolved recursively** to finally form a nested JSON



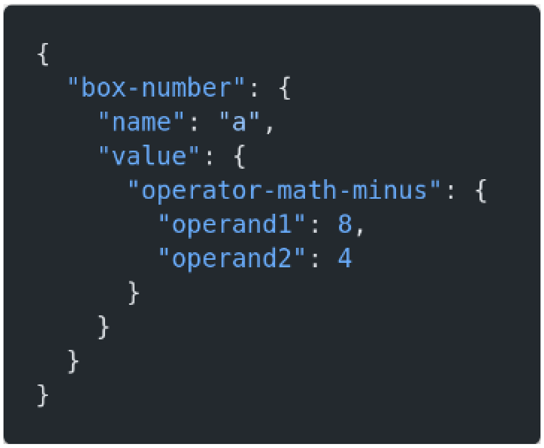
```
{
  "id": "101",
  "argsLength": 1,
  "args": ["105"],
  ...
}
```

```
{
  "id": "105",
  "argsLength": 2,
  "args": ["201", "202"],
  ...
}
```

```
{
  "id": "201",
  "type":"value",
  "value":8
  ...
}
```

```
{
  "id": "202",
  "type":"value",
  "value":4
  ...
}
```

    - After resolving args, this will be the resultant JSON



```
{
  "box-number": {
    "name": "a",
    "value": {
      "operator-math-minus": {
        "operand1": 8,
        "operand2": 4
      }
    }
  }
}
```

- After getting the resolved JSON for the canvas, we can build the program using `@sugarlabs/musicblocks-v4-lib` APIs

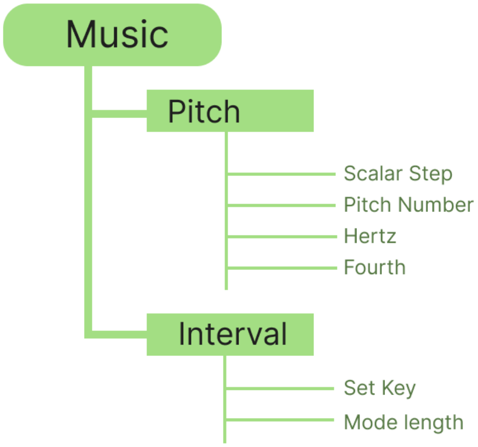
Syntax Tree to Brick Tree:

- Generating a brick tree from a syntax tree will be pretty straightforward.
- We can easily generate a brick tree from the crumbs by resolving nesting in JSON and generating bricks with corresponding IDs and arguments.
- A simple iteration over the crumbs/instructions while keeping track of the first and previous element, and recursively transferring control to the nested instructions, we can get a flat array for the brick tree.

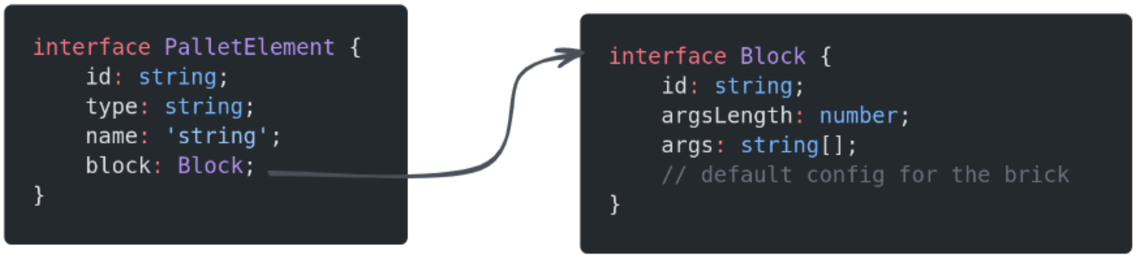
Palette Design

Key points in designing the palette:

- Each class of functions/bricks will be designated a single color.
- These classes can be subdivided further, e.g.:



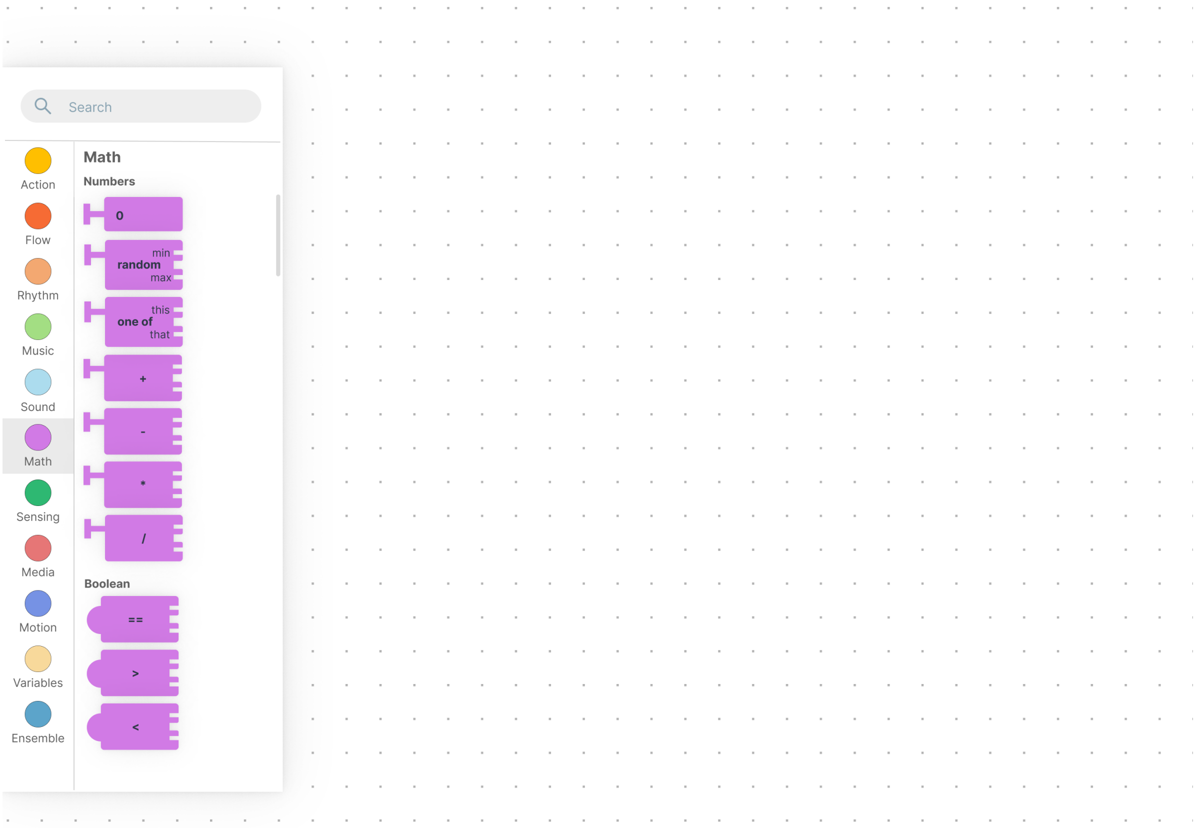
- Blocks can be predefined in a JSON object for each palette component.
- A palette element would look something like this:



```
interface PalletElement {
    id: string;
    type: string;
    name: 'string';
    block: Block;
}
```

```
interface Block {
    id: string;
    argsLength: number;
    args: string[];
    // default config for the brick
}
```

- The state of the palette can be stored in redux, this will allow easy interfacing with the project builder
    - On dragging and dropping a block, dispatch a function to add a block in the redux store
    - Access blocks stored in redux store from the builder.
- Add a search bar on top of the palette
- Design ideas for the palette:



Mockup in Figma

Timeline

Bonding Period

> May 4 - 28

- □ Getting a deeper understanding of the codebase
    - □ Work on small issues
    - □ Learn the logic and flow.

Week 1

> May 29 - June 4

- □ Start with refactoring the prototype
    - □ Resolve some bugs in logic of project-builder
    - □ Make it suitable to be used as a sub-module

Week 2

> June 5 - 11

- □ Start with the integration
    - □ Rewriting/Improve state management of the builder
    - □ Start working on the Wrapper Component

Week 3

> June 12 - 18

- □ Continue with the integration
    - □ Implement functions to tap into builder state
    - □ Making different types of blocks for the brick tree

Week 4

> June 19- 25

- □ Brick Tree To Syntax Tree
    - □ Implement Brick tree to Syntax tree conversion functions

Week 5

> June 25 - July 2

- □ Brick Tree To Syntax Tree
    - □ Testing and debugging

1st Mid-Term Evaluation

> July 10 - 14

Goals for the 1st mid-term evaluation:

- □ Integration of the project builder as a wrapper component
- □ Having basic blocks of musicblocks (Math blocks, action blocks, etc)
- □ Building programs (syntax tree) using a brick tree.

Week 6 - 7

> July 16 - 29

- □ Syntax Tree to Brick Tree
    - □ Implement Syntax to Brick Tree conversion functions
    - □ Testing and debugging of the same

Week 8 - 9

> July 30 - Aug 12

- □ Start working on Palette
    - □ Make a palette component
    - □ Make blocks to include in the palette
    - □ Make a canvas column in the palette to preview blocks and drag and drop blocks from

Week 10

> Aug 13 - 20

- □ Interfacing the Palette with the builder
    - □ Make a redux slice to handle the palette state

2nd Mid-Term Evaluation

> Aug 21 - 28

Goals for the 2nd mid-term evaluation:

- □ A working canvas with an interfaced palette
- □ Palette supporting drag and drop feature to drop elements to the canvas

    - □ Add the currently dragged block to the workspace by getting the cursor position
    - □ Add a search-bar

- □ Working search functionality in the palette.

Week 11 - 12

> Aug 29 - Sep 11

- □ Testing and debugging
    - □ Resolve bugs
    - □ Write basic unit tests for core functionalities.

Week 13 - 14

> Sep 12 - 25

- □ Add block functionalities per block
    - □ Click to trigger for action blocks
    - □ Adding comments or notes to blocks
    - □ Copy/Delete functionality

Week 15 - 16

> Sep 26 - Oct 2

- □ Workspace functionalities and additional features
    - □ Expand/Collapse toggle for flow clamp blocks
    - □ Add buttons for - Zoom in/out, Reset scale, Import/export
    - □ Implementing other extra functionalities

Week 17 - 22

> Oct 3 - Nov 6

- □ Testing and Debugging
    - □ Perform manual end to end and unit testing
    - □ Debug accordingly
    - □ Write tests for core functionalities.

Q. How many hours will you spend each week on your project?

Ans.: I can devote approximately **30-35 hours a week** to GSoC, as it will be my top priority during the summers.

Q. How will you report progress between evaluations?

Ans.: I will make a **weekly report** of what has been accomplished each week. To report progress between evaluations, I will submit these weekly reports and show progress to mentors during a meeting.

Q. Discuss your post-GSoC plans. Will you continue contributing to Sugar Labs after GSOC ends?

Ans.: I plan to **continue contributing to the MusicBlocks** community even after GSoC, as I will have a deeper understanding of the project. If I qualify, I would **like to mentor** for MusicBlocks in future GSoC terms.