Sugar Labs
Google Summer of Code 2023

Project Details:
Name: Music Blocks v4 Project Builder Integration (link)
Length: 350 Hours (Large) (22 weeks)
Mentor: Anindya Kundu
Assisting Mentor: Walter Bender

Student Details:
Full Name: Niloy Sikdar
Email: niloysikdar30@gmail.com
GitHub: niloysikdar
LinkedIn: https://www.linkedin.com/in/niloysikdar
Preferred Language: I'm comfortable and proficient in English both in writing and communicating.
Location: I'm based in Islampur, West Bengal, India
Time Zone: Indian Standard Time (IST) (UTC +05:30)

Previous Open Source Experience:

I have been in a close relationship with open source for the last 2.5 years. Everything started with an open-source program called JWoC (JGEC Winter of Code) while I was in my 2nd year (4th sem) in 2021. Coming from a non-CS background, I didn't have any prior knowledge about Computer Science and programming before entering college. Initially, I tried my hands with Competitive programming but later realized that my interest lay somewhere else.

I started exploring the field of development and open-source and contributed to many exciting projects as a part of JWoC'21. I eventually achieved the 1st Position on the JWOC leaderboard among 600+ participants across different colleges, with 41 successful pull requests in different tech stacks such as Flutter, Firebase, HTML, CSS, JavaScript, etc, after a month of open-source contribution and learning.

Here is the LinkedIn post about the same.

Gradually, I started contributing to other different open-source projects of my own and others in the Web and Mobile domains; including some of the major organizations like **SCoRe Lab, Leopards Lab, MDN, Google, and Fossasia** among many others.

I have also mentored lots of students and beginners to start their open-source journey by conducting different sessions and participating as a Maintainer and Project Mentor at events like LetsGrowMore Summer of Code, GirlScript Education Outreach, and Hacktoberfest 2021, 2022. Also, I was also one of the Lead Organizers of JGEC Winter of Code '22 which helped students to plunge into Open Source contribution and the realm of Software Development. I tried to share knowledge about the world of open-source and pushed the students to contribute to different projects as a part of being the Google Developer Student Club (GDSC) Lead and Speaker at several events, including notable ones like KolkataFOSS.

In 2022, I became a part of **Google Summer of Code (GSoC)** as a Student Developer under the **SCoRe Lab** organization where I worked on solving the problem of building clean, responsive and cross-client compatible emails easily using React.

A few of the noteworthy open-source packages and projects on which I worked are:

1. **React-email:** https://github.com/leopardslab/react-email
   React-based component and utility methods-based lightweight library to provide a common interface for email building that users can install and use to build clean and responsive emails easily.
2. **Neoenv**: https://github.com/niloysikdar/neoenv
   Neoenv is a lightweight, zero-dependency, new-generation module for hassle-free, clean, typed and safe environment variables management for modern developers.

3. **React-awesome-audio:** https://github.com/niloysikdar/react-awesome-audio
   A package with an Optimized and Supercharged React hook to play audio without any DOM element.

4. **JWOC Leaderboard**
   Next.js, React, TypeScript, TailwindCSS, Recoil, MongoDB
   GitHub | Live Link | Live Build
   - Designed and developed the complete leaderboard for an open-source event for 50 projects & 1500+ participants
   - Used Next.js with TypeScript, TailwindCSS for styling, Recoil for state management, MongoDB as the DB
   - GitHub APIs to fetch the data and GitHub Actions to run a Cron Job to update the data at regular intervals

5. **Plaso Connect**
   Tech Stack: Flutter, Firebase, Firestore
   GitHub | Releases | Demo Video
   - Created a mobile application as a one-stop solution with a role-based system where the people requiring blood plasma and oxygen can directly find and contact the donors and healthcare units as per their requirements
   - Integrated Covid API for fetching real-time data, Firestore CRUD operations, animations and page transitions
   - **Got showcased in Google's Dev Library under the Flutter Category**
   - Winner in Health and Safety Track among 3000+ participants at Equinox'21, a national level Hackathon

There are lots of other open-source projects available on my GitHub

Technical Knowledge, Interests, Previous Work:

My general interest lies in building creative and user-centric applications which can solve real-world problems. Recently, I've been exploring the world of build tools and other utility tools for developers, infra and native low-level stuff. I'm well-versed in
Programming Languages: JavaScript, TypeScript, Python, Dart
Libraries/Frameworks: React, Next.js, Zustand, Redux, Recoil, Sass, Material UI, TailwindCSS, Jest, Cypress, Node.js, Prisma, Flutter
Databases: MongoDB, Cloud Firestore, PostgreSQL, MySQL, SQLite, Redis

Tools/Platforms: Git, GitHub, Bitbucket, Heroku, GitHub Actions, Firebase, VS Code, Postman, Agile, Kanban

I have also worked for a few startups to create and scale up their creative products. A few of my experiences are:

- **Front-end Developer Intern | Scenes (previously Avalon Meta) | Nov 2022 - Jan 2023**
  - Worked on a large-scale Discord-like platform that helped to onboard 10+ new B2B customers
  - Fixed bugs and worked on the most requested complex features using React, Redux and Tailwind CSS
  - Optimized the performance by 40% using Code Splitting, Lazy Loading and fixed unwanted multiple renders

- **Technical Lead | TidyForm | Apr, 2022 - June, 2022**
  - Led the Technical team of 4 members and focused on the overall product decisions to solve the problem of creating brandable forms easily using Tidyform

- **DSA and Full Stack Mentor | The 10x Academy | Apr, 2022 - July, 2022**
  - Taking Practice Sessions, Group Discussions and 1:1 mentorship sessions for the students in the domain of DSA and Full Stack Development

- **Software Developer Intern | SAWO Labs | Jul, 2021 – Sep, 2021**
  - Increased the pub points from 80/130 to 130/130 of the sawo flutter package with additional improvements
  - Revamped the whole UI of the website using React and Tailwind CSS with a Lighthouse score of 96
  - Fixed bugs, and added new reusable components to the Client Dashboard and WebSDK using React and Redux
  - Built SAWO sample apps from scratch using React, Vue.js, Node.js and React Native

- **Flutter Developer | Fivefalcon Pvt. Ltd. | Mar, 2021 – Apr, 2021**
  - Developed the interactive and animated UI from scratch of a booking app named Extacy, using Flutter
  - Added Phone, Email and Google authentication using Firebase and Backend features like adding clubs, bars, and parties from the Admin side and showing them on the Client side using Firestore CRUD operations

- **Mobile Application Developer**
    - Designed, Coded and Published 3 Android Applications on Google Play Store in Google Play Store using different technologies like Flutter, Apache Cordova, etc.

Sugar Labs and Me:

I initially came across Sugar Labs and Music Blocks (especially v4) through the GitHub Explore tab. Since I'm a big fan of **TypeScript and React** (mainly performance, optimizations, deep internals, etc.), somehow I found Music Blocks (v4) in my explore section of GitHub. I went through the project, and it seemed quite intriguing to me. I deep-dived into its functionality and code structure and quickly set up the codebase locally. Within a few minutes, it was up and running on my local machine. I found some open issues for the project, and those were mainly related to the build process and infrastructure. As I mentioned earlier that I'm currently falling in love with these topics, so I quickly looked into them and tried to figure out if I could solve them. And voila, I figured out the solutions quickly and after a discussion with the maintainer, I raised my first Pull Request that got merged 🥳

Later on, I deep-dived more into the code, the whole build process, the code structure, and overall features that the app is targeting (since this v4 is the complete rebuilding of the actual Music Blocks, ie. v1 project, there are and will be lots of the same features).

Here is a detailed summary of the work and contributions which I did prior to the GSoC proposal period:

Discussions in which I participated:
https://github.com/sugarlabs/musicblocks-v4/issues/288#issuecomment-1399603559

Issues Raised:

- **[Bug] Script failing to generate the i18n and assets sizes while building (#302)**
    - Major issues for not getting the proper output data for `dist/stats.json` in the `win32` platform. I pointed out the exact problem with proper details and also proposed one solution.

- **[Bug] Assets sizes collection from build (#317)**
    - The issue with the current approach in collecting asset sizes from build output leads to inaccuracies; adjustments are needed to properly obtain actual sizes and incorporate hashed image paths for better loading percentage calculations.

Merged Pull Requests:

- **fix(build): stats script for win32 (#307)**
  - Fixed the stats data generating script for the win32 platform that solves issue #302

- **fix(stats): fix assets stats collection (#318)**
  - Used native fs module to get all the asset files (png, jpg, jpeg, gif, svg, mp3, ogg, wav, xml for now) and their sizes, then write the dist/stats.json file with the complete hashed path of the assets with their sizes in bytes to solve the issue #317. This also helped to fix the loading percentage issue related to #288.

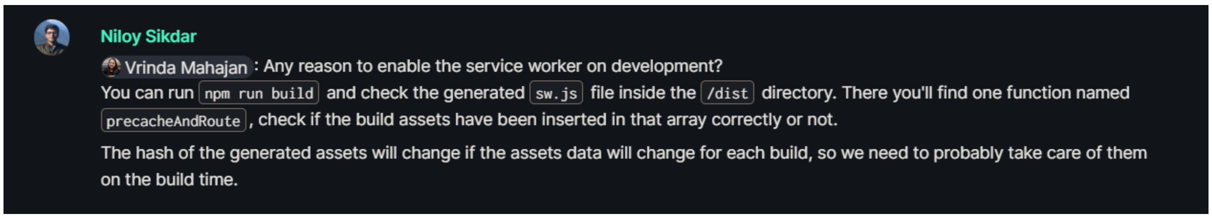
- **feat(splash): sync splash progress with the sizes of items (#319)**
  - Syncing and updating the splash progress with the stats data using item sizes instead of item counts.

Through these code contributions, I can strongly say that I have got a good grasp and understanding of the overall structure, build process, and internals of the codebase. I'm now quite confident to promptly get the idea of which file needs to be edited or where should we include any new file with the exact specific content to fix any specific bug or push some new features.

Communication with the Community:

As an open-source contributor and maintainer, I firmly believe that helping other new contributors to start and help them contribute to the issues is as important as our own contributions. That's why I also try to become as much active as possible in the community to help others to contribute.

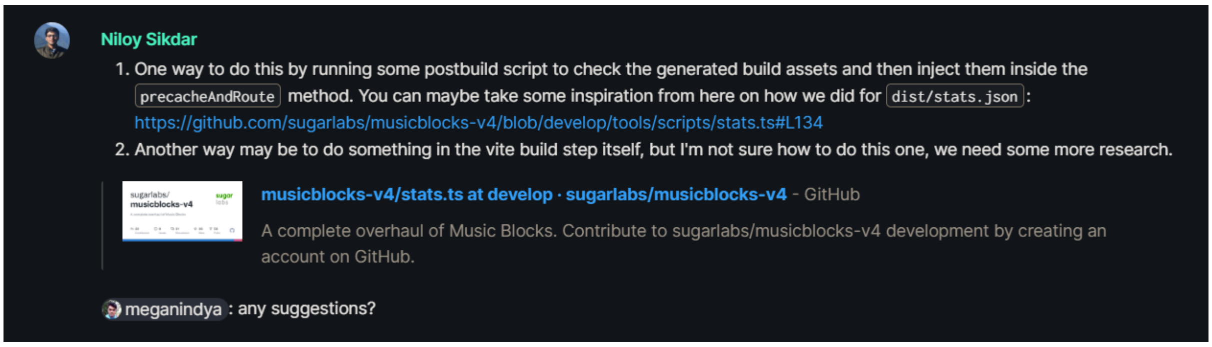
A few snapshots from the Element public conversation are here:



Niloy Sikdar

Vrinda Mahajan : Any reason to enable the service worker on development?
You can run `npm run build` and check the generated `sw.js` file inside the `/dist` directory. There you'll find one function named `precacheAndRoute`, check if the build assets have been inserted in that array correctly or not.

The hash of the generated assets will change if the assets data will change for each build, so we need to probably take care of them on the build time.



Niloy Sikdar

1. One way to do this by running some postbuild script to check the generated build assets and then inject them inside the `precacheAndRoute` method. You can maybe take some inspiration from here on how we did for `dist/stats.json`:
https://github.com/sugarlabs/musicblocks-v4/blob/develop/tools/scripts/stats.ts#L134
2. Another way may be to do something in the vite build step itself, but I'm not sure how to do this one, we need some more research.

musicblocks-v4/stats.ts at develop · sugarlabs/musicblocks-v4 - GitHub
A complete overhaul of Music Blocks. Contribute to sugarlabs/musicblocks-v4 development by creating an account on GitHub.

meganindya : any suggestions?

Apart from these, I had some detailed conversations with my mentor Anindya Kundu through GitHub comments and Element chat. I also had the opportunity to have a conversation with other mentors Walter Bender and Devin Ulibarri in a Google Meet.

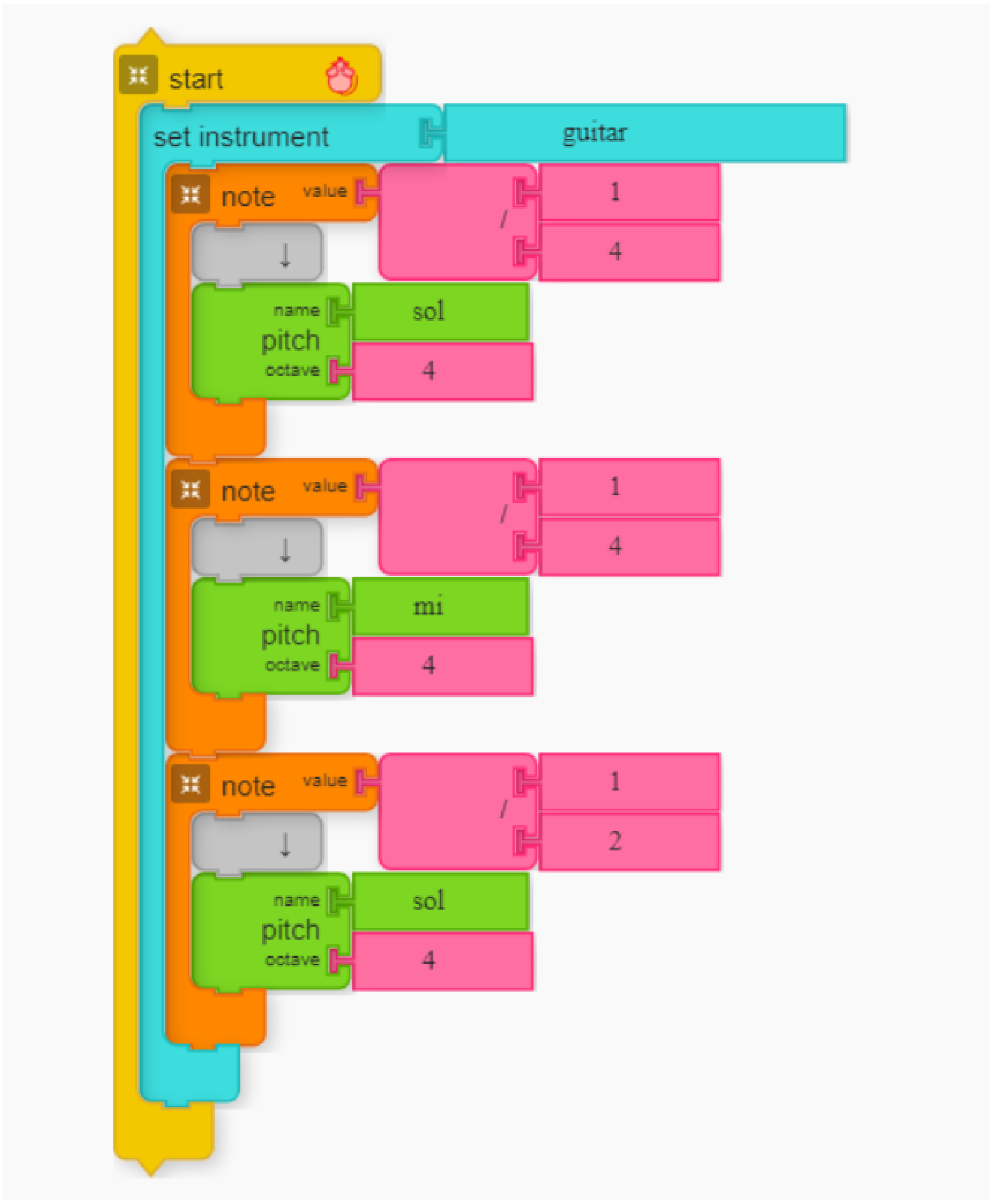
Experience so far:
As I mentioned earlier, I initially came across Music Blocks through the GitHub Explore tab. But now, the concept and gist of the project, contributing to the project, having properly detailed discussions with the mentor, and engaging with the community - all of these things have deeply attached me to this project and organization. I'm quite excited and looking forward to continuing my work and also helping other developers to start contributing to the project.

Project Details:

Introduction:
To get started with the context, the **Music Blocks v4** project is basically "a complete overhaul of Music Blocks", which got originated from the "Turtle Blocks" project. Now there is already one complete end-to-end implementation of Music Blocks in pure native Vanilla JavaScript without any front-end framework (eg. Angular, Vue) or libraries (eg. React). But that actually started long 8 years ago (in 2015, primarily by Yash Khandelwal @khandelwalYash as a part of GSoC 2015) when the front-end tooling and the whole ecosystem weren't that much modern and up to the standard. Now in today's time, the whole front-end development has changed with so many new modern tools, frameworks and libraries in the market, as well as different mental models in terms of UI rendering for building web applications including Client-Side Rendering (CSR), Server-Side Rendering (SSR), Static-Site Generation (SSG), Incremental Static Regeneration (ISR), etc. Maintaining the old project is also becoming tough due to the lack of modularization, and the performance is also not up to that mark comparing today's applications' performances which are built using modern technologies. That's why, the "Music Blocks" project needed to start from scratch using a better application architecture, different modern tools (like Vite, ESLint, Prettier, Docker, Jest, Cypress, etc), better languages (TypeScript for end-to-end type-safety and static type-checking, Sass for styling) and libraries (React for UI rendering) in terms to improve the overall developer and user experience, maintainability, and performance. That's why the v4 project got started and it's still a Work In Progress (WIP).

Till now, there are lots of things that got ported to v4, but most of the major features are still missing. The whole project's progress can be found here.

One of the main features of Music Blocks is the **"Project Builder"**. The Project Builder is the graphical blocks manager module that can be used to create Music Blocks programs. Now there are different types of blocks and each of them represents a different type of functionality, like the Start block, Rhythm block, Note block, Pitch block, different kinds of instrument blocks (eg. Drum), and to pick them, etc. All of these blocks can be found here for Music Blocks v1 project (I'm referring to the old one as v1 and the new one as v4). These blocks are completely interactive, which means we can drag and drop different blocks, click them to take actions or open any context menu, or do some other stuff like delete any specific block. Below is an example from the v1 project of how different types of blocks can be combined together in a group to form one complete Musical Project:



Here, we can clearly see that some blocks are combined together visually to create music just by using some drag and drop, clicks and providing values for other settings.

But the drawback for the v4 project is, this is missing right now inside the application. So, there's a high need to implement this inside the v4 project. **I would like to implement this feature end-to-end so that users can start creating different musical patterns using this "Project Builder".**

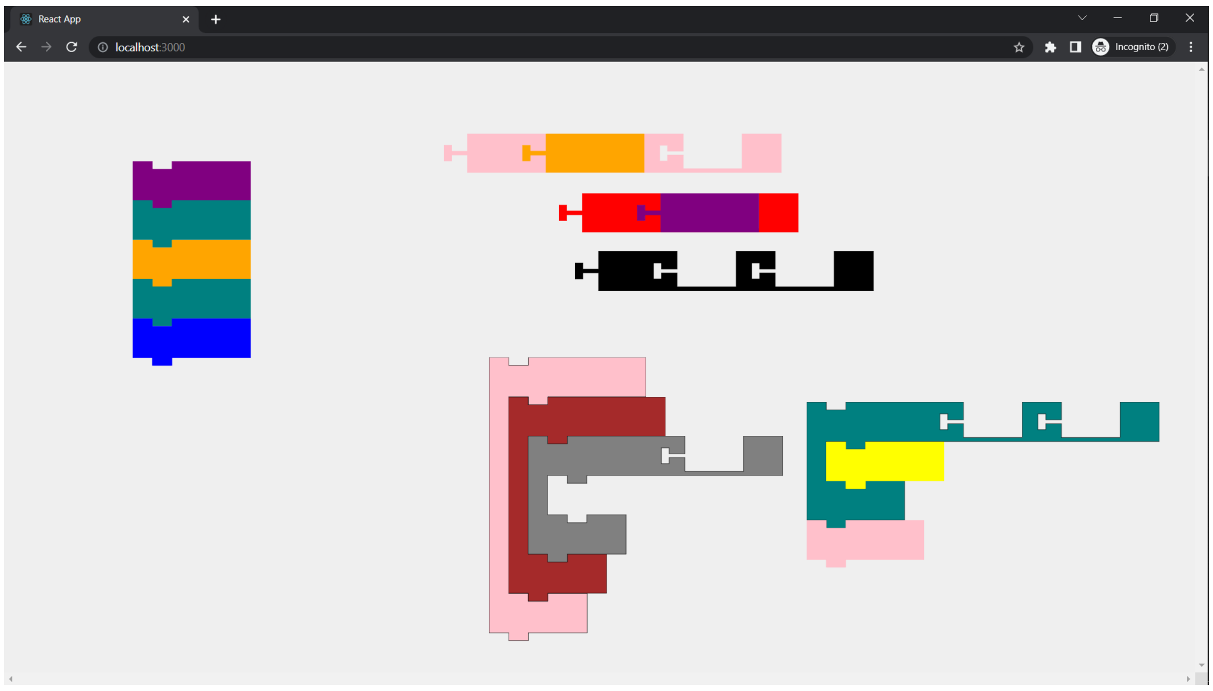
Impact on Sugar Labs:

The mission of Sugar Labs is to stimulate learning. Music Blocks is a Visual Programming Language and collection of manipulative tools for exploring musical and mathematical concepts in an integrative and fun way and it is one of the most active, famous, and important projects under Sugar Labs. As we are completely porting and migrating the old VanillaJs application to the modern v4 one, we need all of the major features of the old project also into the new one. And if we consider the "Project Builder", that's basically **one of the most important features and parts of the entire application**. While there can be the option of using text-based languages (eg. JavaScript) for generating music, as we have in the old one but providing a block-based visual approach to generate the music is more important because it provides a smoother learning curve for children, and beginners as different components in the code could be visualized as UI elements and easily interacted with. That's why I feel this is one of the most important projects for this year's Google Summer of Code, and completing this project will surely add a huge impact on the Music Blocks' users and the whole Sugar Labs community. Being an experienced developer and open-source contributor, I want to take the end-to-end responsibility to build the feature for the v4 application from ground zero and help the entire Sugar Labs community to start playing with it.

Project Goals and Implementations:

The primary objective is to build the new Project Builder Framework for Music Blocks (v4). There is already one existing prototype in musicblocks-v4-builder-framework which Saurabh Raj created as a part of Google Summer of Code 2021. He created the **"Learning Bricks - A Novel Blocks Framework for Visual Programming Languages"** which is less opinionated, easily customizable, lightweight, and performant. Learning Bricks was originally developed for Music Blocks v4 but could be easily customized to cater to the needs of any project. Now as a part of this year's GSoC, I would like to integrate "Learning Bricks" to create the "Project Builder" framework for Music Blocks v4 by taking some reference from the already existing prototype and making sure not to completely copy things and optimize the code as per the need.

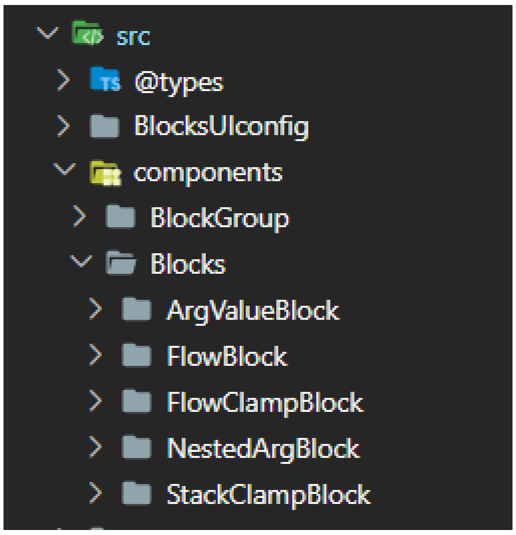
One of the primary objectives of the project is to deeply understand the code and the logic of the existing prototype and how this can be integrated with the Music Blocks project. I have already set up the prototype's codebase locally on my machine, fixed some minor bugs, and it's completely up and running. Here is one screenshot of the project running on `*localhost:3000*`.



I played with the blocks to form different groups and deep-dived into the code to debug how things are actually working. Here is a detailed breakdown of the implementation process:

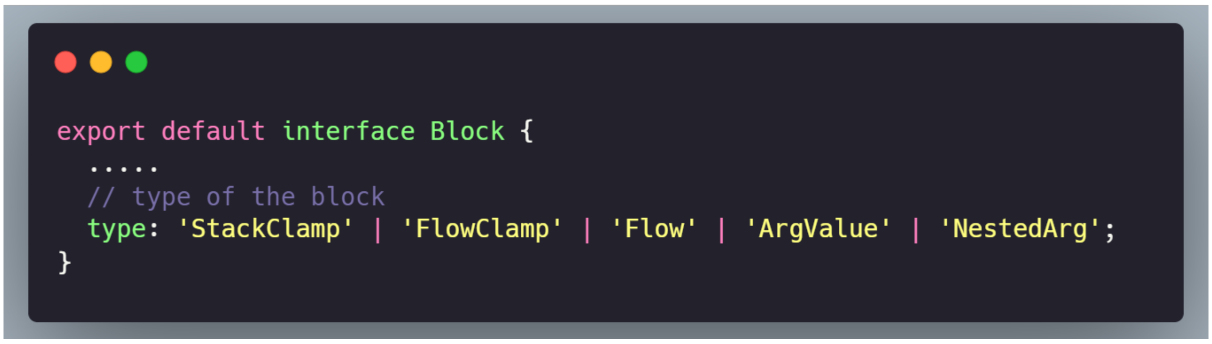
- **The Building Blocks:**

As I mentioned earlier, the Project Builder consists of different types of blocks which can be grouped together in the scene to form a musical sequence. Now, these blocks are highly important and can be implemented from the already existing prototype. There are already different types of blocks inside the `src/components/Blocks` folder.



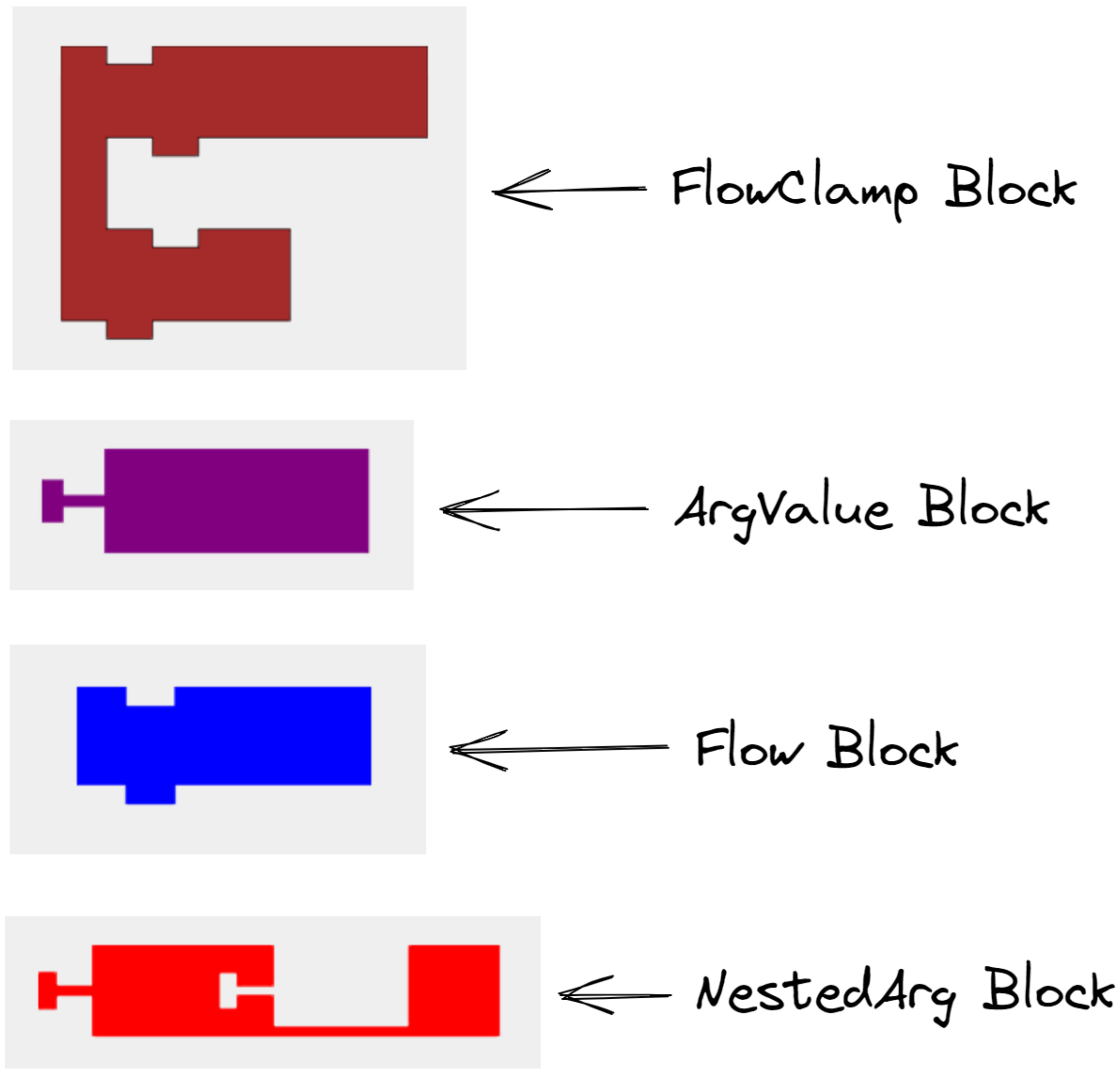
```
src
  @types
  BlocksUIconfig
  components
    BlockGroup
    Blocks
      ArgValueBlock
      FlowBlock
      FlowClampBlock
      NestedArgBlock
      StackClampBlock
```

As we can see here, there are different types of blocks which represent different shapes:



```
export default interface Block {
  .....
  // type of the block
  type: 'StackClamp' | 'FlowClamp' | 'Flow' | 'ArgValue' | 'NestedArg';
}
```

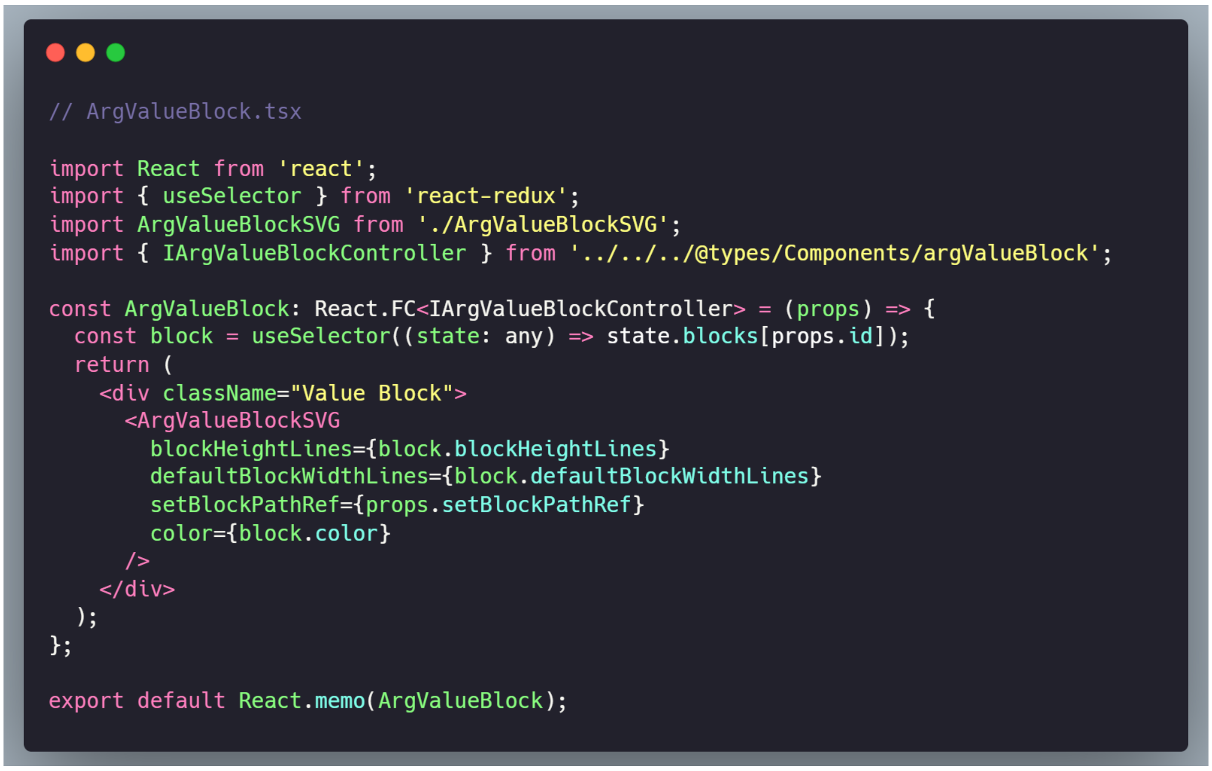
Here are the examples of how each of these blocks looks like:



Each of these blocks is just an SVG which is getting rendered dynamically based on the props' values, also they're using React.memo to memoize the expensive computations and to avoid unwanted multiple re-renders.

Here is an example of the `**ArgValueBlock**` component:

Each of the blocks has its own id, we're getting the block data from the store based on the block id and using it to pass those data to the `ArgValueBlockSVG` component which is returning the actual SVG with those computed values.



```
// ArgValueBlock.tsx

import React from 'react';
import { useSelector } from 'react-redux';
import ArgValueBlockSVG from './ArgValueBlockSVG';
import { IArgValueBlockController } from '../../../@types/Components/argValueBlock';

const ArgValueBlock: React.FC<IArgValueBlockController> = (props) => {
  const block = useSelector((state: any) => state.blocks[props.id]);
  return (
    <div className="Value Block">
      <ArgValueBlockSVG
        blockHeightLines={block.blockHeightLines}
        defaultBlockWidthLines={block.defaultBlockWidthLines}
        setBlockPathRef={props.setBlockPathRef}
        color={block.color}
      />
    </div>
  );
};

export default React.memo(ArgValueBlock);
```

Here is an example of how the `ArgValueBlockSVG` component is returning dynamic SVG based on the passed props. There are a few specific reasons for using SVG (Scalable Vector Graphics) instead of PNG (Portable Network Graphics) for rendering the blocks. There are several advantages when we need to manipulate or alter the image in web applications and these blocks need some manipulation based on their position and neighbouring blocks' positions.

The advantages are:

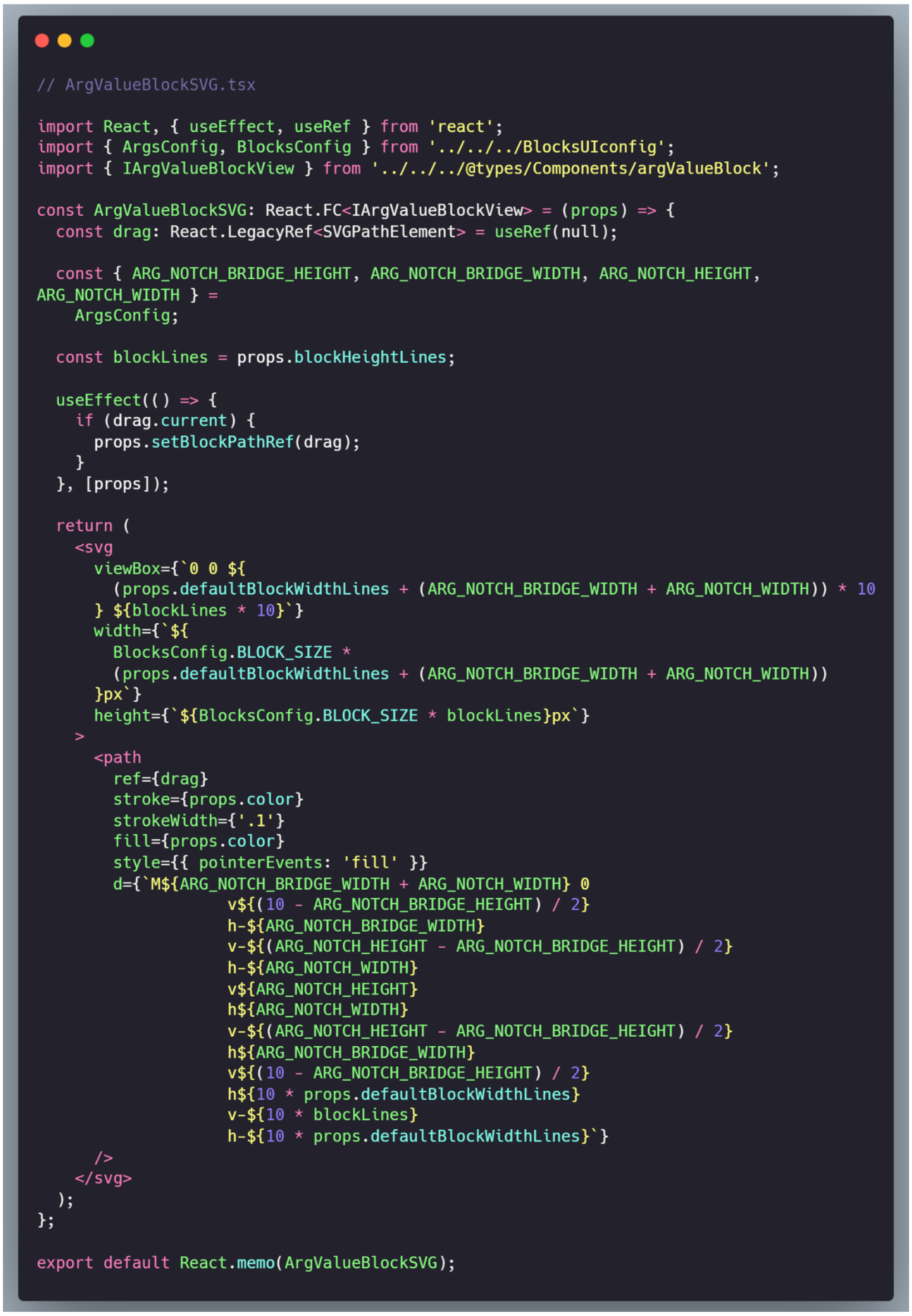
1. *Scalability:* SVGs offer scalable, dynamic graphics without loss of quality, making them ideal for responsive designs, while PNGs are raster images that suffer from pixelation when scaled.

2. *Dynamic Manipulation*: SVGs enable direct HTML embedding and dynamic manipulation using JavaScript and CSS, whereas PNGs are static images requiring external libraries or new image generation for editing.

3. *Smaller File Sizes:* SVGs typically have smaller file sizes than PNGs for simple shapes, leading to faster load times and better web performance.

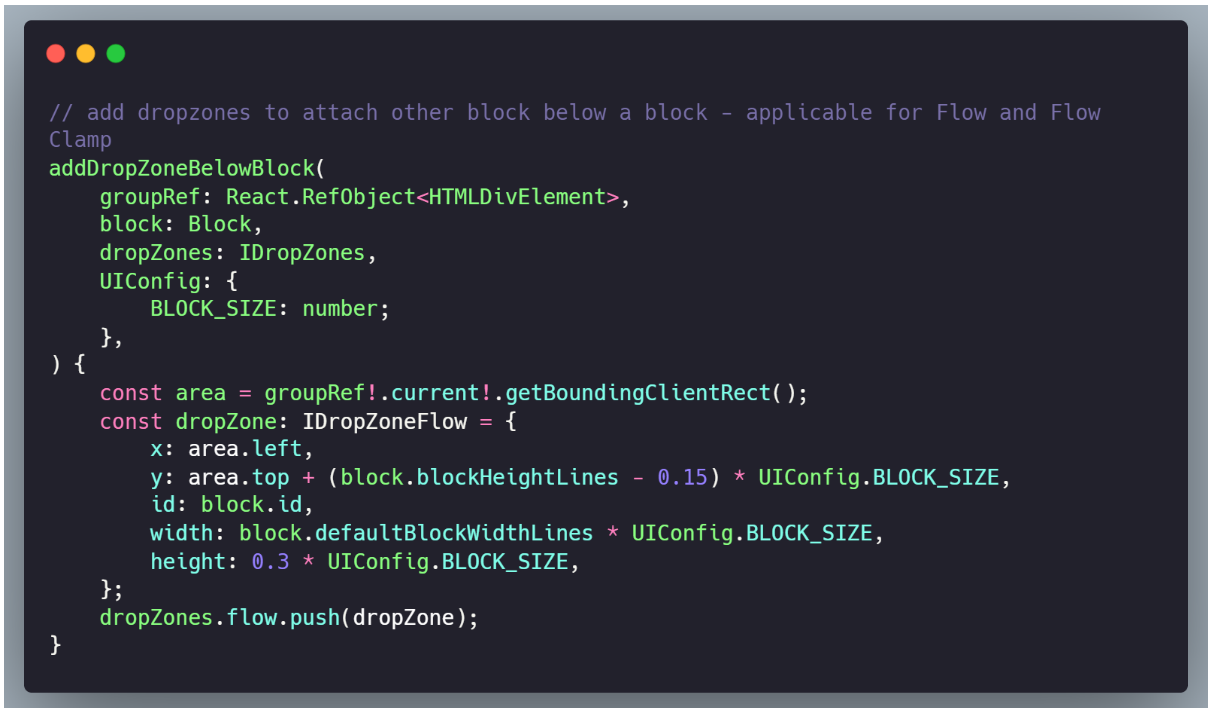
4. *Accessibility*: SVGs, being text-based markup, offer better accessibility for screen readers and assistive technologies, aiding users with disabilities.

The blocks are grouped to form the `BlockGroup`. Now this `BlockGroup` component is pretty interesting because not all the blocks are draggable separately. If we drag the block from the bottom, then it will be separated, and the same will also happen



```tsx
// ArgValueBlockSVG.tsx

import React, { useEffect, useRef } from 'react';
import { ArgsConfig, BlocksConfig } from '../../../BlocksUIconfig';
import { IArgValueBlockView } from '../../../@types/Components/argValueBlock';

const ArgValueBlockSVG: React.FC<IArgValueBlockView> = (props) => {
  const drag: React.LegacyRef<SVGPathElement> = useRef(null);

  const { ARG_NOTCH_BRIDGE_HEIGHT, ARG_NOTCH_BRIDGE_WIDTH, ARG_NOTCH_HEIGHT,
ARG_NOTCH_WIDTH } =
    ArgsConfig;

  const blockLines = props.blockHeightLines;

  useEffect(() => {
    if (drag.current) {
      props.setBlockPathRef(drag);
    }
  }, [props]);

  return (
    <svg
      viewBox={`0 0 ${
        (props.defaultBlockWidthLines + (ARG_NOTCH_BRIDGE_WIDTH + ARG_NOTCH_WIDTH)) * 10
      } ${blockLines * 10}`}
      width={`${
        BlocksConfig.BLOCK_SIZE *
        (props.defaultBlockWidthLines + (ARG_NOTCH_BRIDGE_WIDTH + ARG_NOTCH_WIDTH))
      }px`}
      height={`${BlocksConfig.BLOCK_SIZE * blockLines}px`}
    >
      <path
        ref={drag}
        stroke={props.color}
        strokeWidth={'.1'}
        fill={props.color}
        style={{ pointerEvents: 'fill' }}
        d={`M${ARG_NOTCH_BRIDGE_WIDTH + ARG_NOTCH_WIDTH} 0
                    v${(10 - ARG_NOTCH_BRIDGE_HEIGHT) / 2}
                    h-${ARG_NOTCH_BRIDGE_WIDTH}
                    v-${(ARG_NOTCH_HEIGHT - ARG_NOTCH_BRIDGE_HEIGHT) / 2}
                    h-${ARG_NOTCH_WIDTH}
                    v${ARG_NOTCH_HEIGHT}
                    h${ARG_NOTCH_WIDTH}
                    v-${(ARG_NOTCH_HEIGHT - ARG_NOTCH_BRIDGE_HEIGHT) / 2}
                    h${ARG_NOTCH_BRIDGE_WIDTH}
                    v${(10 - ARG_NOTCH_BRIDGE_HEIGHT) / 2}
                    h${10 * props.defaultBlockWidthLines}
                    v-${10 * blockLines}
                    h-${10 * props.defaultBlockWidthLines}`}
      />
    </svg>
  );
};

export default React.memo(ArgValueBlockSVG);
```

for the top one. But if we can drag any block from the stack of blocks placed vertically, the block group will detach from the top, which means it will be separated from the below blocks and the top blocks will be separated. The same thing is also true for the horizontally aligned blocks, if we try to drag any block from the middle, then it will be separated from the left. The same logic is also applicable to the "Project Builder" inside "Music Blocks" where we can detach and drag different block groups to create different musical arrangements.



```
// add dropzones to attach other block below a block - applicable for Flow and Flow
Clamp
addDropZoneBelowBlock(
    groupRef: React.RefObject<HTMLDivElement>,
    block: Block,
    dropZones: IDropZones,
    UIConfig: {
        BLOCK_SIZE: number;
    },
) {
    const area = groupRef!.current!.getBoundingClientRect();
    const dropZone: IDropZoneFlow = {
        x: area.left,
        y: area.top + (block.blockHeightLines - 0.15) * UIConfig.BLOCK_SIZE,
        id: block.id,
        width: block.defaultBlockWidthLines * UIConfig.BLOCK_SIZE,
        height: 0.3 * UIConfig.BLOCK_SIZE,
    };
    dropZones.flow.push(dropZone);
}
```

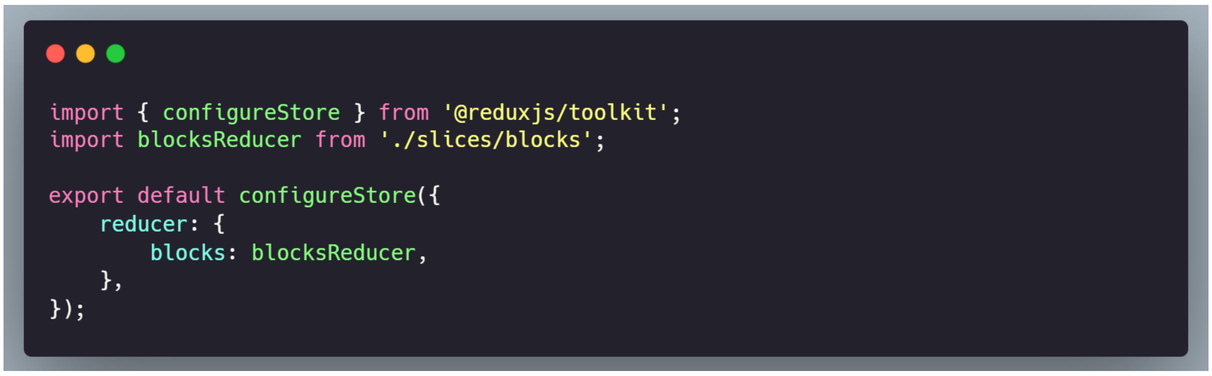
The same thing is also applicable to the other types of blocks. The implementation can be found here inside the `src/components/BlockGroup/BlockGroupController.ts` and `src/DropZones/DropZonesController.ts` files.

- **Managing the state of the different blocks:**

There are lots of blocks that are placed here and there in the scene. But now managing the state (position and other important data) for each of the blocks is pretty difficult. That's why we need the help of any global state management solution. Here in the case of the prototype, Redux Toolkit has been used to create and initialize a global store to keep a track of each of the block's data.

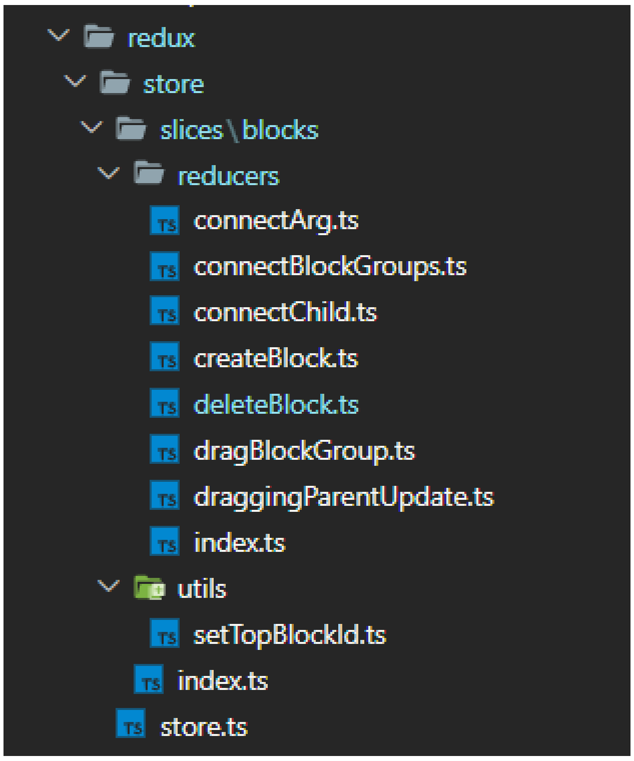
The blocks and components **dispatch** different **actions**, which eventually reach the **reducers**. Now the reducers modify the **Central Store** which holds all the state of the application. Now they trigger **Subscriptions** which then notify the components and update the UI based on the updated state values.

Here, we're configuring a global store with all the reducers with the help of `@reduxjs/toolkit`.



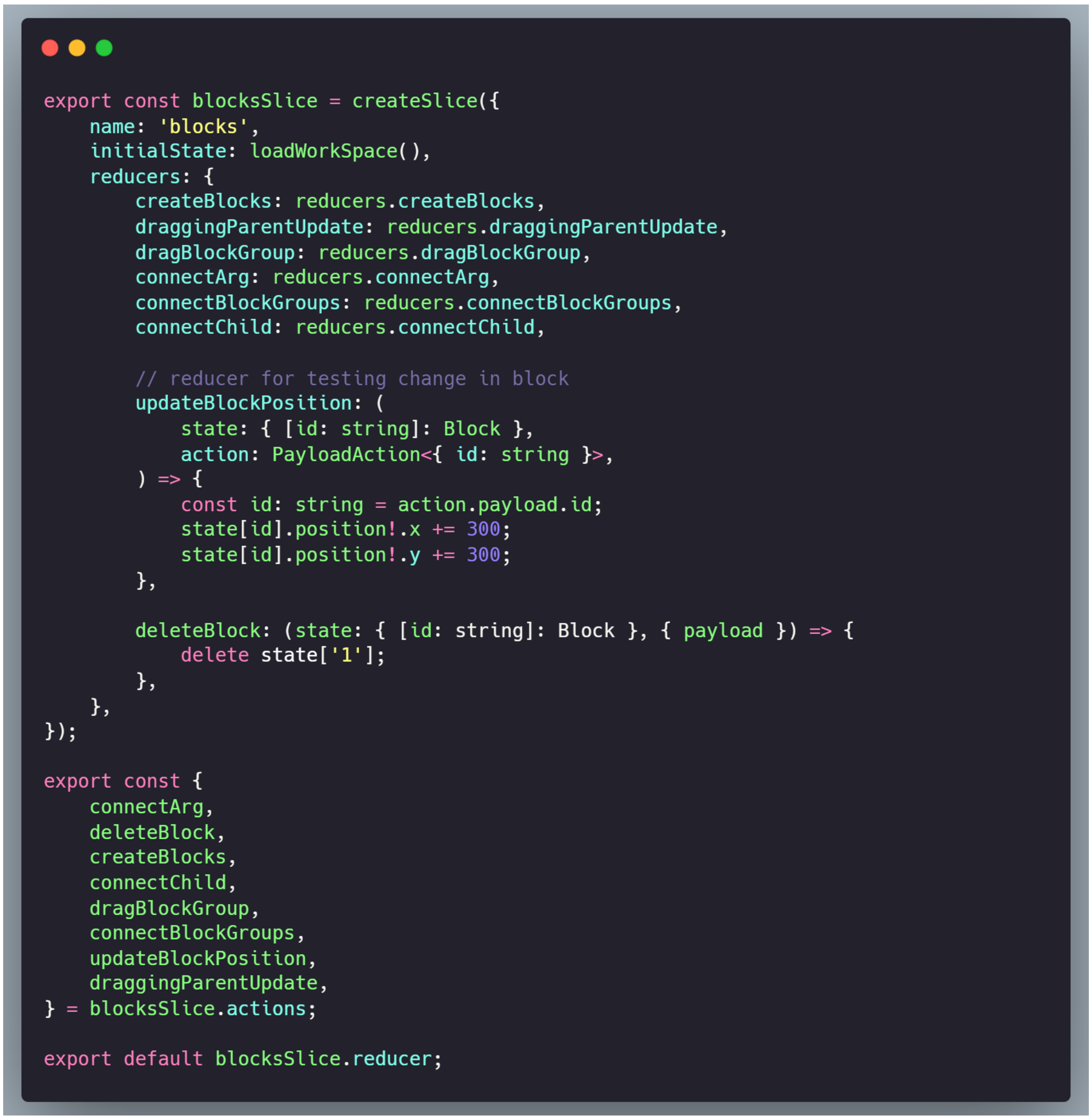
```
import { configureStore } from '@reduxjs/toolkit';
import blocksReducer from './slices/blocks';

export default configureStore({
    reducer: {
        blocks: blocksReducer,
    },
});
```

Now we're defining different reducers and actions. Reducers are the functions which are responsible to modify the store based on the action payload. In the prototype, we have different reducers that are responsible to perform different operations in the store.



```
redux
  store
    slices\blocks
      reducers
        connectArg.ts
        connectBlockGroups.ts
        connectChild.ts
        createBlock.ts
        deleteBlock.ts
        dragBlockGroup.ts
        draggingParentUpdate.ts
        index.ts
      utils
        setTopBlockId.ts
      index.ts
    store.ts
```

Using Redux Toolkit, we can easily get the actions by defining the reducers. Also, we need to pass the `initialState` of the store. In this case, there is one demoWorkspace that loads initially. The blocks which are defined there will get rendered based on their initial positions and other data, and then the values get updated directly after drag and drop.



```
export const blocksSlice = createSlice({
    name: 'blocks',
    initialState: loadWorkSpace(),
    reducers: {
        createBlocks: reducers.createBlocks,
        draggingParentUpdate: reducers.draggingParentUpdate,
        dragBlockGroup: reducers.dragBlockGroup,
        connectArg: reducers.connectArg,
        connectBlockGroups: reducers.connectBlockGroups,
        connectChild: reducers.connectChild,

        // reducer for testing change in block
        updateBlockPosition: (
            state: { [id: string]: Block },
            action: PayloadAction<{ id: string }>,
        ) => {
            const id: string = action.payload.id;
            state[id].position!.x += 300;
            state[id].position!.y += 300;
        },

        deleteBlock: (state: { [id: string]: Block }, { payload }) => {
            delete state['1'];
        },
    },
});

export const {
    connectArg,
    deleteBlock,
    createBlocks,
    connectChild,
    dragBlockGroup,
    connectBlockGroups,
    updateBlockPosition,
    draggingParentUpdate,
} = blocksSlice.actions;

export default blocksSlice.reducer;
```

We can see the list of reducers and actions for directly manipulating the glocal store data for the different blocks:

- connectArg
- deleteBlock
- createBlocks
- connectChild
- dragBlockGroup
- connectBlockGroups
- updateBlockPosition
- draggingParentUpdate

Based on the updated store state and values, the blocks and block groups will render on the UI.

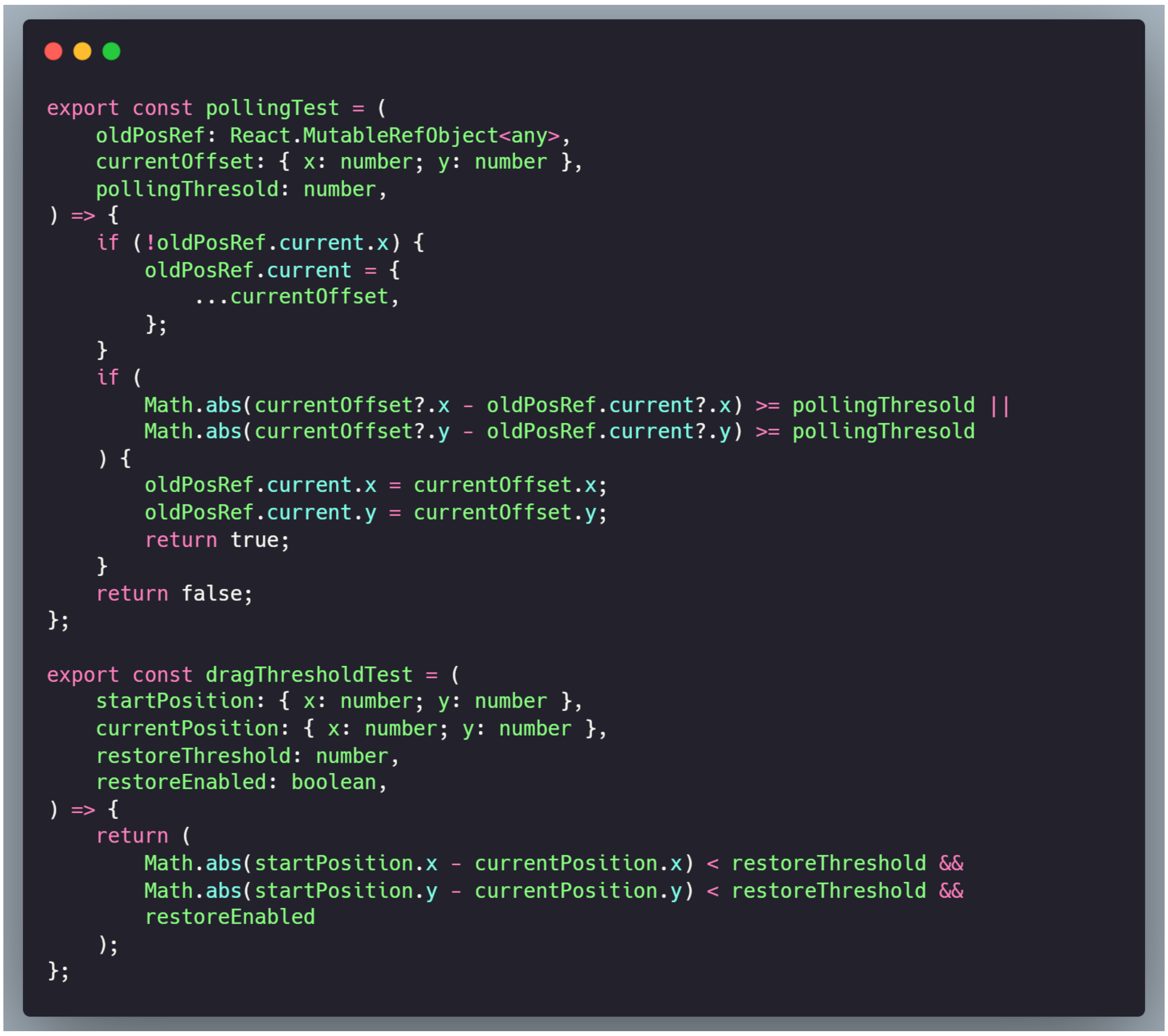
- **Drag and Drop:**

I have mentioned the **“drag and drop”** mechanism multiple times, and now it's time to discuss its internal logic and implementation. Drag and drop is one of the most important parts of the project. Blocks need to be able to drag and drop, and we need to take actions programmatically based on the drag events. Users can drag the blocks and place them in different positions to combine the blocks to form the melody. To handle this, D3 has been used. Based on the drag events, we’re executing different functions. We have 3 different event types for the drag events: “start”, “drag” and “end”.

When we’re dragging any block, first, we’re updating the position of the current block, also making the “z-index” to some higher value (1000 has been used in the prototype) so that it will be placed at the top of the other blocks.

After the drag end, first, we’re doing some calculations based on the start position and the end position. We’re checking if the difference between the start and end position is beyond the threshold, then only update the newer position of the block. That means, if we’re dragging the block just a little bit, which isn’t crossing the threshold value, then the block’s position will be restored to the initial position. That will help us to prevent updating the block positions after each and every small drag and will ensure only updating the positions after a significant amount to drag which will improve the overall performance.

All of the dragging logic can be found here: `src/utils/dragging.ts`



```
export const pollingTest = (
    oldPosRef: React.MutableRefObject<any>,
    currentOffset: { x: number; y: number },
    pollingThresold: number,
) => {
    if (!oldPosRef.current.x) {
        oldPosRef.current = {
            ...currentOffset,
        };
    }
    if (
        Math.abs(currentOffset?.x - oldPosRef.current?.x) >= pollingThresold ||
        Math.abs(currentOffset?.y - oldPosRef.current?.y) >= pollingThresold
    ) {
        oldPosRef.current.x = currentOffset.x;
        oldPosRef.current.y = currentOffset.y;
        return true;
    }
    return false;
};

export const dragThresholdTest = (
    startPosition: { x: number; y: number },
    currentPosition: { x: number; y: number },
    restoreThreshold: number,
    restoreEnabled: boolean,
) => {
    return (
        Math.abs(startPosition.x - currentPosition.x) < restoreThreshold &&
        Math.abs(startPosition.y - currentPosition.y) < restoreThreshold &&
        restoreEnabled
    );
};
```

- **Collision detection between blocks:**

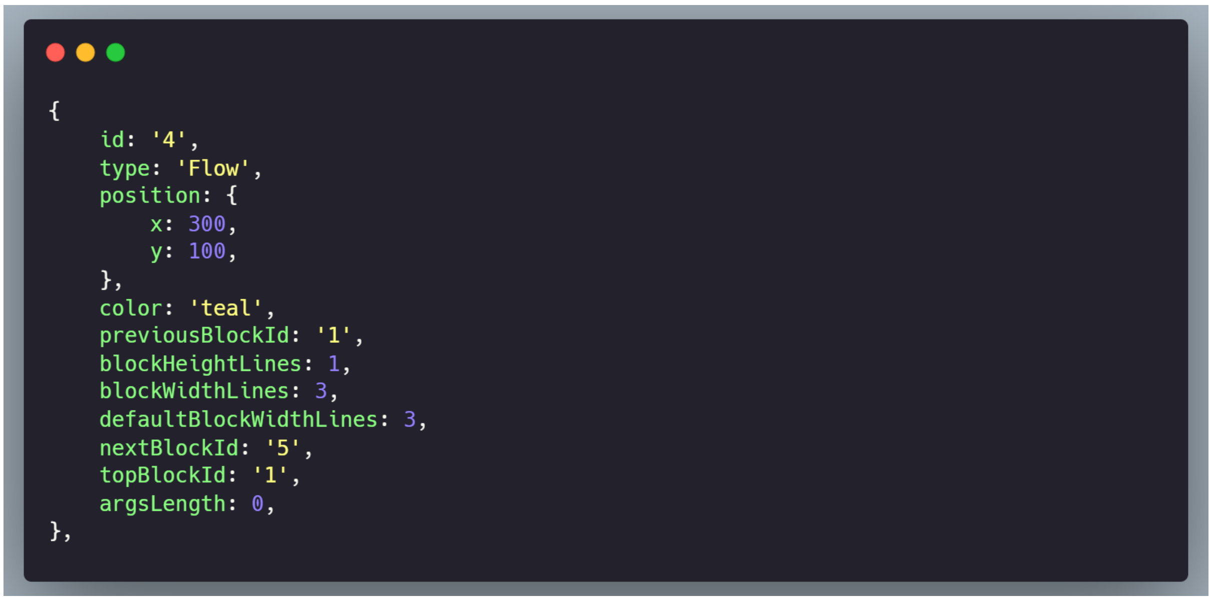
Detection of the collision between the different blocks after the drag end is the toughest and trickiest part. After the drag end and while dropping the block, first, we need to check for the `pollingTest` and `dragThresholdTest`, if they'll return true then find if the block which we have dragged is colliding with any of the other existing blocks. If colliding, then we need to do calculations based on that and update the parent block and the global store based on the latest data. We need to inject the block to the exact position, and to the exact slot where it can be fitted only.

To detect this collision, the brute force way would be to check the positions of every two nodes, which means checking each point against every other point. But that'll result in a time complexity of **O(N^2)**, where N is the number of data points. This approach quickly becomes computationally expensive as the number of data points increases, making it impractical for a large number of blocks. That's why we're using the **Quadtree** algorithm. The quadtree algorithm recursively subdivides a 2D space into quadrants until each region contains a single data point or is empty, allowing for efficient storage and searching of data points in the space. The quadtree algorithm provides a more efficient approach for searching and inserting data points in a 2D space. In general, the quadtree algorithm provides efficient searching and insertion operations with a time complexity of **O(log N)** in the average case, where N is the number of data points.

To use this quadtree algorithm, we're using the quadtree-lib package inside the prototype instead of writing and building from scratch. We're also checking the collisions and getting the data both vertically and horizontally using the helper methods from the package.

The code implementation can be found here: src/DropZones/DropZones.ts

After colliding with the other blocks and if it has the chance to fit into the slot of the other block (we can also make it closer to the slot without actually putting it in the exact position and it will be fitted automatically if has the scope), then the previousBlockId, nextBlockId, topBlockId, and argsLength will get updated for the blocks. This will also cause changes in the argWidth and blockLines for the blocks, and the overall shape, height and width will get updated to adapt the design of the newly created **Block Group**.


```
{
    id: '4',
    type: 'Flow',
    position: {
        x: 300,
        y: 100,
    },
    color: 'teal',
    previousBlockId: '1',
    blockHeightLines: 1,
    blockWidthLines: 3,
    defaultBlockWidthLines: 3,
    nextBlockId: '5',
    topBlockId: '1',
    argsLength: 0,
},
```

The code for the updateArgWidths method can be found here: src/utils/argWidths.ts
Since ArgValue Block doesn't need any changes in the width in any of the cases, that's why we're checking the type of the block and if it's equal to "ArgValue", then do nothing and simply return the previous state values.

The code for the updateBlockLines method can be found here: src/utils/blockLines.ts

The main usage for the drag and then collision detection is used here inside the src/components/BlockGroup/BlockGroup.tsx component (because one single block is ideally also a Block Group).

Objectives of the Project:

As I have already added the detailed breakdown of the prototype code, here are the major objectives on which I want to work as a part of Google Summer of Code 2023.

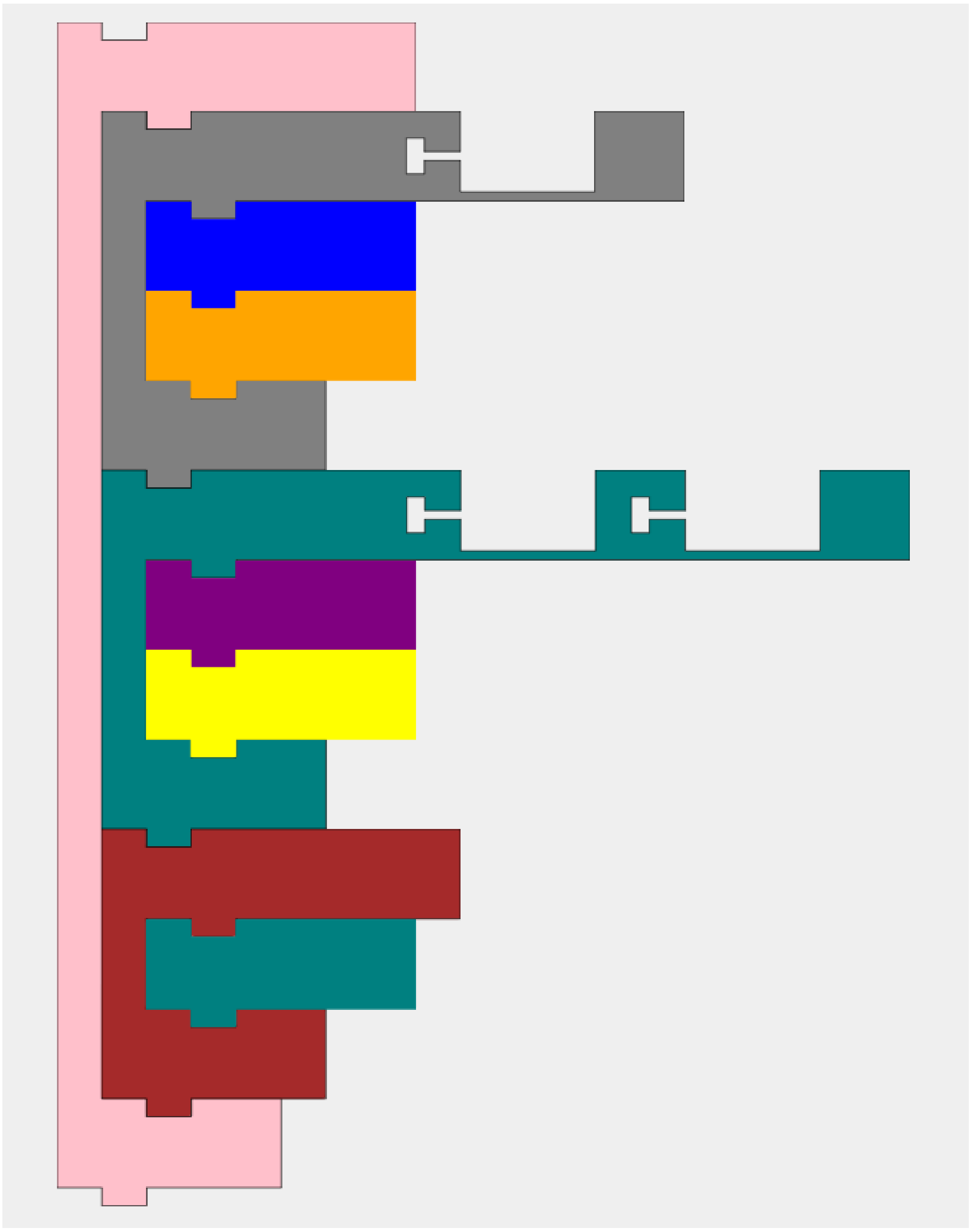
- **Refactor the Prototype code:**

The first thing before integrating it into the Music Blocks v4 project, is to refactor the already existing code. We need to optimize the code and make it more modular and generic so that it can be easily integrated into the Music Blocks v4 project.

- **Integrate it in "Music Blocks v4":**

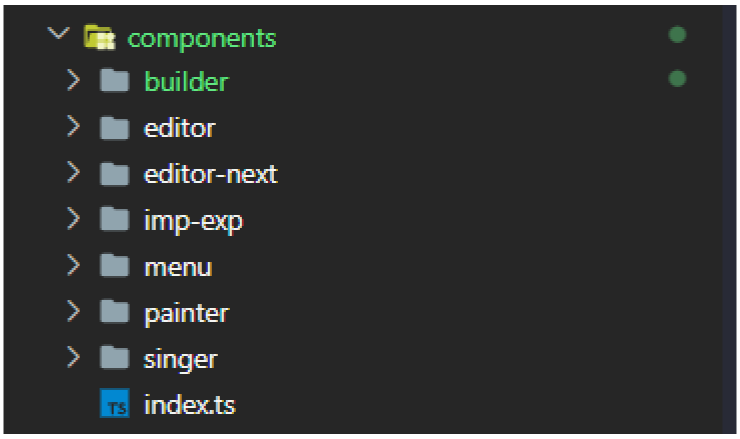
The next and most important task will be definitely to integrate it into Music Blocks v4 project. The "Learning Bricks" will be used as the Musical Blocks (ie. Start block, Rhythm block, Note block, Pitch block, etc) for the Music Blocks project. As the structure and shape of the bricks of the prototype are also similar to the blocks of the Music Blocks project, this can be easily integrated into the project with just some minor modifications.

Here is one example from the prototype to create an almost similar grouping structure for Music Blocks.



- **Create a wrapper component Project Builder (builder) in musicblocks-v4:**

I'll create a standalone `builder` component inside the `src/components` folder following the same code, file, and folder structure as other components (ie. editor, menu, painter, singer, etc.). I'll also try to decouple the view (UI) from the business logic as much as possible. This will contain all the logic and view for the “Project Builder” component. The size of the component will be added automatically to the generated *dist/stats.json* file inside *“modules”* while building the app.



```
components
  builder
  editor
  editor-next
  imp-exp
  menu
  painter
  singer
  index.ts
```

- **Communication with the Programming Framework:**

I’ll also create some utility methods inside the `builder` wrapper component which will help us to communicate with the Specification and Syntax Tree APIs of the Programming Framework. Now, this should be a two-way communication, where we can generate the array containing multiple blocks with their details to render to the UI, from the Programng Framework’s APIs. As well as we should be also able to generate the exact JS syntax structure, from the array of Blocks, that we need for communication with the APIs and methods.

The structure and type of data which we need for generating the different blocks into the UI are here inside the demoWorkspace.

We can also take references from the tests about the structure of the data which is required for communicating with the Specification and Syntax Tree APIs of the Programming Framework.

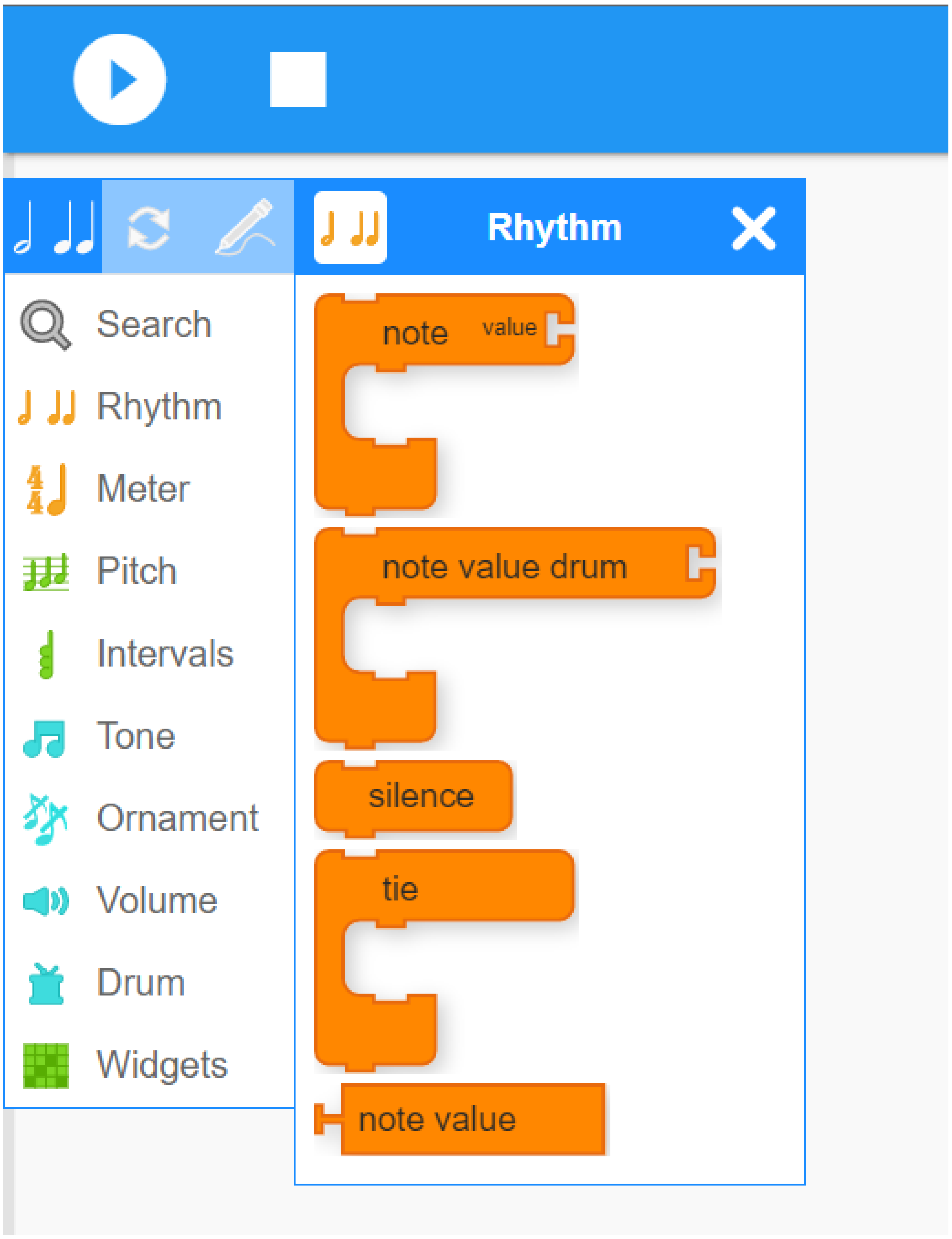
Tests for “Specification”: src/syntax/specification/index.spec.ts
Tests for “Tree”: src/syntax/tree/index.spec.ts

- **Create a Palette (palette) component:**

So now, we are clear on how blocks can be used for creating different block groups in the scene. But the question is how can we add a new block of a specific type?

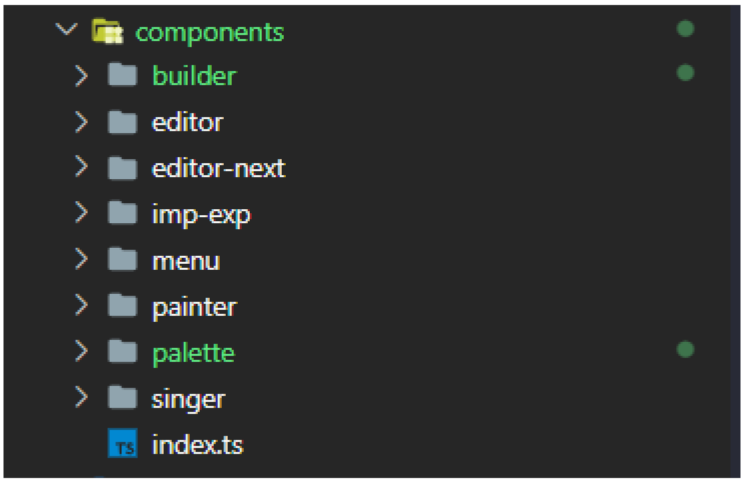
For that, we need a common universal Palette from where we can drag any specific block to the scene.

Here is an example of the palette which contains different types of blocks for different use cases. It can be found on the left side of the current (v1) Music Blocks application.



I'll create another separate `palette` component inside the `src/components` folder which will be responsible for rendering and handling the logic for this. I'll follow the same principles for this component that I've mentioned for the "builder" component.

To implement this, we can load some hardcoded blocks for each type of option and use cases, similar to what we are doing right now with demoWorkspace data inside the prototype to load the initial blocks to the scene. We can detach the block from the palette which has been dragged into the scene by updating the values and detecting collision with the already existing blocks on the scene. Also, we need to refill the block into the palette which has been detached, by again simply adding the same block with the same values and state into the specific palette array.

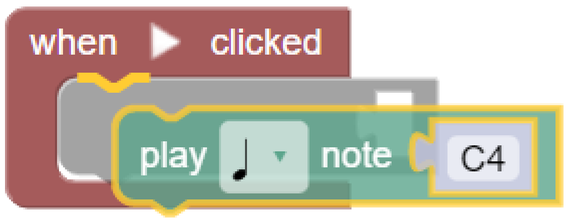


```
components
  builder
  editor
  editor-next
  imp-exp
  menu
  painter
  palette
  singer
  index.ts
```

Other objectives on which I'll work as a part of GSoC 2023 are:

- **Realtime preview of the collision part**

Right now the prototype doesn't support the real-time preview of the collisions, which means that users won't get the idea of with which block the current block is getting collided and fitting into the slot. We can drag and after the drop, the block will get fitted into the other block's slot without any preview. To make the user experience better we can show a preview of the fitting on the drag state itself without dropping the block. And after the drop, it will get fitted to the preview slot as usual. We can take some references from Blockly Music Game and implement the same thing in our application.



We can run the collision detection function for the drag event (while the user is dragging any block), and show a container with a border of the preview. But the problem is if we run this expensive method for each and every drag event so many times, it might cause lag and affect the overall performance and smoothness of the application.

One way to optimize this is by using some debounced condition just like “debounced search” which will enforce the detection function not to be called again until a certain amount of time has passed without it being called (like 200ms).
But we can first try normally and then execute some performance tests, and if we’ll face any kind of lag or performance issues, then we can follow some other fixes.

- **Create missing functions and blocks:**

There are a few blocks of different shapes missing in the prototype, I’ll create and add those missing blocks in the main application. Additionally, I'll also implement the missing methods and functions which are necessary for the project.

- **Add text to the SVGs:**

If we carefully observe the Music Blocks’ blocks, there are labels on each of the blocks which represent the type and work of the blocks. But this thing is missing in the prototype’s code. These blocks are SVGs, so we just need to show some text inside the SVGs, that we can pass dynamically through the props.

Here are some of the good solutions from Stack Overflow on how to show text inside of SVGs. We can follow any one of the methods which seemed fine and efficient.

- **Add Tests:**

Testing is also a very important part of software development. That’s why I’ll add the tests for the components and the methods simultaneously. We have Jest and Cypress already set up on the v4 project for testing purposes. Jest is used for Unit Testing and Cypress is used for End to End testing. Also, the CI has been already configured with GitHub Actions to run the tests on the pipeline. I’ll try to improve the test coverage by adding the necessary tests where needed.

- **Add support for Mobile devices and use alternate libraries:**

Right now, we're using the d3.drag.on("start"/"drag"/"end") event listeners for listening to the drag events triggered by the mouse. But the problem is these aren't completely compatible with the touch events on mobile devices. That's why we need to find some solutions (for example from Stack Overflow) to improve the user experience. Also, we need a lot of work on the UI part to make it mobile-friendly. I'll try to work on these issues as a part of GSoC if I'll get enough time before the end of the program. Otherwise, I can still work on these improvements after the GSoC timeline as normal open-source contributions and work.

Also, we can use the d3-drag library instead of the native d3 library, as we're only using the drag functionalities from d3.

Also, we can maybe use Zustand instead of Redux Toolkit as it is smaller, fast, scalable, and easy to use (with lesser boilerplate code) by using simplified flux principles.

- **Add support for Husky and lint-staged:**

The codebase for the Music Blocks v4 project is well structured with proper integrations with the different tools and packages (eg. ESLint, Prettier, CI/CD, etc.). However, we can also use Husky for initializing git hooks with lint-staged which will help us to maintain proper commit rules and prevent the developers from committing code with improper checks and commit messages. One example of the whole setup can be found in one of my open-source projects here.

Overall Tech Stack for the project:

The project heavily depends on the following tech stack:

React v18 with Functional Components based approach and hooks, TypeScript, JavaScript DOM API for the DOM manipulation based on the different events, Canvas API to render the different blocks on the screen, some state management solutions (eg. Redux), different JavaScript/TypeScript libraries.

Timeline:

Period	Tasks
Pre-GSoC Period *Till May 4*	● More deep dive into the prototype's codebase and contribute to the v4 project. ● Try and test different alternatives for the libraries and approaches. ● Research on optimization techniques and other features.
Community Bonding *May 4 - May 28* *(3 weeks and 4 days)*	● Discussion on the project started ● Decide the best approach to handle different events and other methods ● Discuss with the mentor if anything we are missing before the coding period. ● Deciding the final tasks and onboarding them to the task board to keep proper track of them.
Coding Phase 1 *May 29 - July 10* *(6 weeks)*	● Add support for husky and lint-staged (if necessary). ● Refactor the prototype code as much as possible. ● Integrate it in "Music Blocks v4" ● Create the wrapper component Project Builder (builder) in musicblocks-v4 ● Updating tests and documentation (with JSDoc comments for the code) simultaneously ● Bug fixes
Phase 1 Evaluation *July 10 - July 14*	● Complete the backlogs (if any) ● Implement the feedback received
Coding Phase 2 (extended deadline for large projects) *July 14 - Nov 6* *(16 weeks and 4 days)*	● Create missing Functions and Blocks ● Create a Palette component ● Communication with the Programming Framework ● Add text/labels to the SVGs ● Realtime preview of the collision part ● Adding more tests for the views and methods ● Update documentation (with proper JSDoc comments) accordingly ● Add more support for Mobile Devices (if time permits) ● More Bug fixes

Final Mentor Evaluation *Nov 6 - Nov 13*	● Final date for mentors to submit evaluations for GSoC contributor projects with extended deadlines

Availability:

I’m focusing to dedicate 30-35 hours/week to the project and will mostly be active between Thursday - Sunday, between 11 AM - 7 PM IST. As I'm currently in my final year of engineering, there’s a high chance of the final semester examinations being held towards the end of May or the first week of June as per the academic calendar. So maybe I’ll be a little bit inactive for the first 1-1.5 weeks during the initial coding phase (if the semester takes place at that time). After that, I won’t have any college academic engagements and I can devote more time to the project.

Progress Report:

It has always been my wish for a long time to release my own blog site, but for many reasons, it wasn't happening. I will try to code my own blog site and write detailed blogs about the updates, my overall GSoC experience, and about the Sugar Labs community and its projects. Also, I already have one Medium account with the username niloysikdar that I also used last year to document my GSoC progress. So if the own blog site won’t happen, then I’ll publish the blogs to the medium itself.

Post-GSoC Plans:

Once GSoC concludes, I will look into the issues and the PRs raised by the other developers and will also look into new issues. Adding some more features and making the project more robust and feature-rich will be my primary goal. I want to maintain the Music Blocks v4 for the long term and want to become a lifelong member of the Sugar Labs community to help people in the field of open-source. Apart from the Music Blocks v4 project, I’m also planning to explore and contribute to the other projects under Sugar Labs (ie. Programming framework of musicblocks-v4, Sugarizer, Sugar, Music Blocks v1, etc.) after the GSoC period.

Conclusion:

Having shared a fairly comprehensive overview of my project and its planned execution, my principal plans for GSoC 2023 would be to expand upon my deep understanding of the project that I have gained through my practical experience from prior contributions and research.

Regarding the tech stack, I’m completely familiar with all the technologies which are needed for this project. I have been working extensively with **React**, **Redux** (as well as other state management solutions), and **TypeScript** (even pushing the type-safe paradigm by building different packages for the developers). A fun fact is, I was a big fan of **Canvas API** and Three.js and even created a game with it. Having completed GSoC last year, I’m also familiar with the entire workflow and other technicalities of the event. Being an experienced and responsible developer with a diverse open-source background having in-depth knowledge about the project and prototype codebases and how things can be implemented for the project, I can clearly say that I’ll be able to complete this project with higher expectations within the mentioned timeline, and I want to take the end-to-end responsibility to implement all the features that are extremely crucial and valuable for the future of the project and take Music Blocks v4 to the next level.

Oh, I forgot to mention that apart from programming, I am a foodie, a part-time chef on weekends, and a keen traveller with a taste for Bollywood music.

Thanks for reading, and I’m expecting and looking forward to utilizing this summer to the fullest by working under the mentorship of talented mentors and with the community.