

Sugarizer Word Puzzle and Chart activities

Proposal : Gsoc 2023

Basic Details:

Name:	Utkarsh Siddhpura
Email:	utkarshsiddhpura09876@gmail.com
Github:	UtkarshSiddhpura ↗
Languages:	English, Hindi
Location:	Gujarat, India
Timezone:	India Standard Time (UTC +5:30)
Work Hours:	9-20(UTC) Flexible

I am committed to complete project during the summer. I am available to work on the project whenever needed and will make it my top priority to ensure that it is finished on time.

Open Source:

- My journey began with the Hacktoberfest event, where I was introduced to the concept of contributing to open source Projects. I completed it.
- I made several contributions to different projects, including this: [ztm](#)
- While these contributions were small in scope, they were valuable experience.

Sugarizer:




- I was researching organizations for the Gsoc program and came across SugarLabs, the organization behind Sugarizer, a free/libre learning platform. I was immediately intrigued by the idea of creating an educational platform that is accessible to children around the world, regardless of their socioeconomic background.
- I started contributing to Sugarizer by identifying\fixing bugs in the codebase. After spending some time exploring,I have taken the time to study the codebase of the Exerciser activity, including its interaction with Sugarizer.
- Pull Request: [Sugarizer Pull Requests](#)
- Issues: [Sugarizer issues](#)
- Exerciser Issues: [Exerciser issues](#) | [PR](#)
- [Sugarizer activity development tutorial.](#)

Project Details:

1.What am i making ?

Chart Activity: The Chart activity can help children to visualize and understand complex data sets in a more engaging/interactive way, making it easier for them to learn and explore new concepts.

Word Puzzle template: The new Word Puzzle template in the Exerciser activity allows teachers to quickly create custom word puzzles during lessons, giving learners an interactive way to practice vocabulary.

- I believe that I would be a **great fit** for this project, as I have a strong track record of developing successful projects using similar technologies. My experience of building Games using JS and Web apps using React and Redux has given me a deep understanding of the tools and frameworks which are also used in the Sugarizer & Exerciser activity.
- **List of some Projects uses technologies**: HTML, CSS, JS, React, react-router & redux.
 1. [E-commerce SPA](#)  | [code](#) template anyone can use for their products.
 - Authentication, Payment, SPA, Persist data, Orders, Responsive, etc.
 2. [2048](#) , [Tetris](#) ,
 3. and many more projects see on github.

Libs/Framework i know: Sugar-web/core, Activities(js/vue), Vue, react-ecosystem, react-intl, chart.js, etc.

2.How will the Project impact Sugar Labs ?

The addition Chart activity and Word Puzzle in Sugarizer can have a positive impact on Sugar Labs by providing learners and educators with more tools to engage with the platform. Chart activity can help learners visualize and analyze data in a creative and interactive way, while Word Puzzle can improve their vocabulary and spelling skills. This can lead to more widespread adoption of Sugarizer, and greater engagement and satisfaction among users.

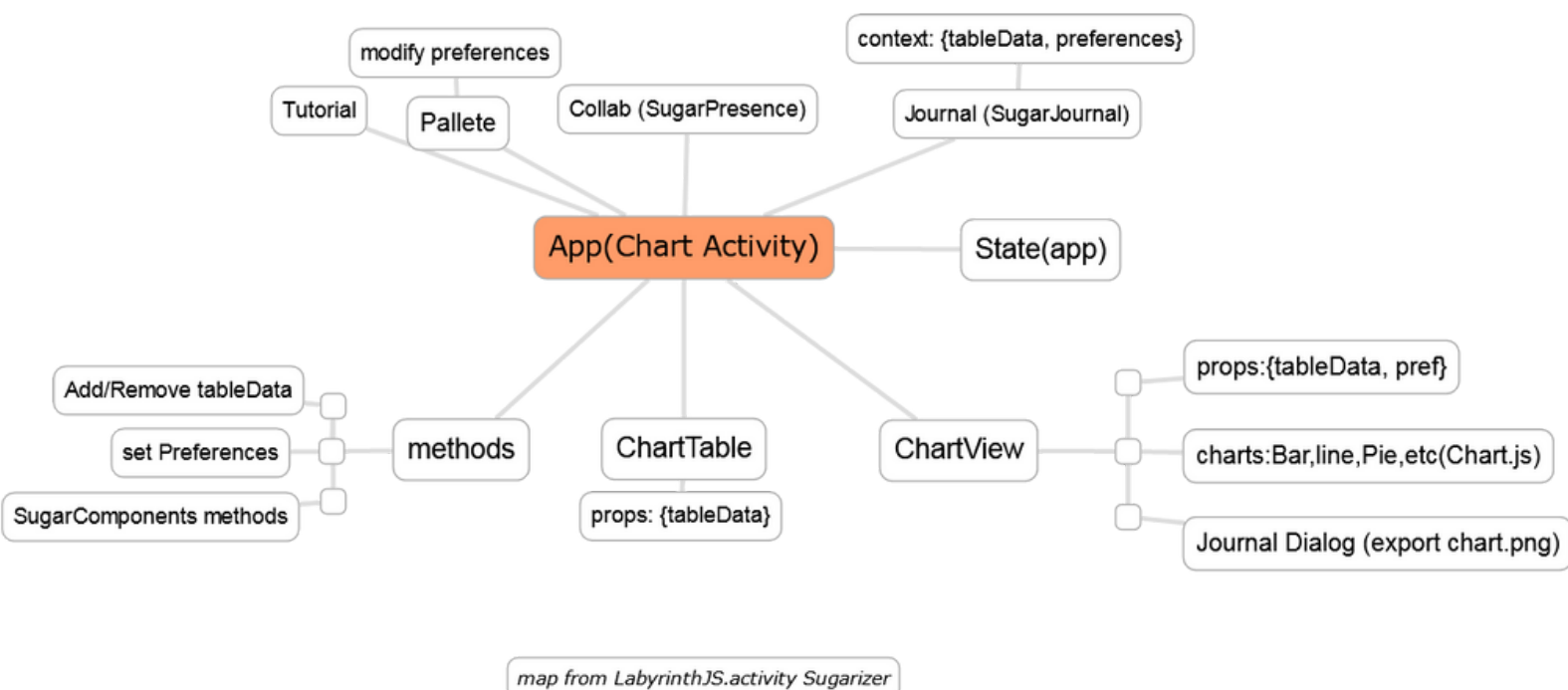
3.Technologies for Project

Chart Activity: HTML, CSS, Vue.js, Sugar-web(Vue components for Presence, Journal, Localization, Toolbar, Tutorial, etc), Chart.js, require.js.

Word Puzzle: HTML, CSS, React, redux, react-router, react-intl, intro.js & Sugar-web.

Implementation:

1. Chart Activity:



```
State(app): {
  tabularData: [
    { label: x1, value: y1 },
    { label: x2, value: y2 },
    // and so on...
  ],
  selectedField: 0,
  activityTitle: "",
  preferences: { ... }
}
```

```
preferences: {
  config: {..} ,
  options: {
    heading: {
      fontSize: _,
      fontFamily: _,
      // .....
    },
    chartType: "line",
  }
}
```

Features :

1. Add data with the "add-data" icon:



- add sugar toolitem

```
<sugar-toolitem id="add-data" title="Add " @click="addData"></sugar-toolitem>
```

- addData method will add empty label & Value fields with focus on.

```
addData() {
  this.tabularData.push({label: "", y: 0});
  this.$nextTick(function () {
    this.$refs.input[lastIdx].focus();
  })
}
```

- tableView will render all the values on screen and update them on edit.

```
<div v-for="(data, i) in tabularData" class="field-container..">
  <input v-model:value="data.label" ref="input" type="text" />
  <input v-model:value="data.value" type="number" />
</div>
```

2. Remove data with the "remove-data" icon:



- To remove data, we'll use selectedField (default to last input field index.)
- on clicking any field -> set value -> index of clicked field.

- on Icon click will delete the item stored in tabularData.

```
tabularData.splice(selectedIndex, 1);
```

- modifying state will make require UI changes except chart for now.

3. Chart View on canvas:

component :

```
var ChartView = {
  template: `
    <div class="canvas-container">
      <canvas ref="canvas" class="chart-canvas"></canvas>
    </div>
  `,
  props: ["tabularData"],
  //.....
```

on mount: Draw the chart from labels and values.

```
mounted: function () {
  this.chart = new Chart(context, {
    type: "bar", //type, color hard coded
    data: {
      datasets: [
        {
          data: this.values,
          borderColor: "#36A2EB",
          backgroundColor: "#9BD0F5",
        },
      ],
      labels: this.labels,
    },
    options: {..},
    scales: {
      y: {
        beginAtZero: true,
      },
    },
    .....
```

- Labels & Values are computed :

```
computed: {
  labels() {
    return this.tabularData.map((data) => data.label);
  },
  values() {
    return this.tabularData.map((data) => data.value);
  },
},
```

- Labels & Values also Watched to update chart dynamically

```
watch: {
  labels() {
    this.chart.data.labels = this.labels;
    this.chart.update();
  },
  //same for values
}
```

4. Horizontal/Vertical bar charts, line charts, and pie charts:



- Type of charts can be set using preferences.type/chartType.
- Add sugar-toolitem for each type & onClick setType(type).

```
<sugar-toolitem id="..." :title="..." @click="setType('line')"></sugar-toolitem>
```

- to improve, we could use constant CHART_TYPES (global) which contains all types and add sugar-toolitem by iterating over it.
- In setType func set -> pref.type = CHART_TYPES[argPassed]

```
<chartview class=".." :tabular-data="tabularData" :pref="pref"></chartview>
```

- set this.pref.type to this.chart.config.type & update chart.

5. Palette Config:



- Helps to set Horizontal/Vertical labels with label Colors.



```
<sugar-toolitem
  id="config-button"
  title="..."
  palette-file="js/palettes/configpalette.js"
  palette-class="configpalette"
></sugar-toolitem>
```

- I'll implement a configpalette.html file for UI which contains 2 buttons
- 2 input form fields with x→ and y ↑ Icons
- (id: xlabel-color/input & ylabel-color/input)
- For Functionality in configpalette.js:
- dependencies: [graphics/palette.js](#), [graphics/colorpalette.js](#), configpalette.html.
- Implementing custom events :

```
this.configChange= document.createEvent('CustomEvent');
this.configChange.initCustomEvent('config-change', true, true, { config: {..}
});
```

- Color Palettes to select colors & Dispatching events

```
XlabelPalette = new colorpalette.ColorPalette(xlabel-color);
XlabelPalette.setColor('rgb(0, 0, 0)');

XlabelPalette.addEventListener('colorChange', function(e) {
  that.configChange.config.colors.x = e.detail.color;
  that.getPalette().dispatchEvent(that.configChange);
});

// same for YlabelPalette
```

- Above code is making color palettes for x & y, on colorChange will dispatch configChange event.
- Similarly for xlabelInput and ylabelInput, onchange will dispatch event with config.labels.x = e.target.value.
- Now we've updated preferences.config which is reactive and will update the chartview.

6. Palette Font :

- Setup fontPalette.js, fontPalette.html & toolitem like above.
- html template will contain a font chooser with font size increment/decrement icon also color chooser.

- For font chooser ("arial", "verdana"...), use `textpalette.js` like it's in [Labyrinth](#)
- `addEventListener` for "font-increment/decrement" buttons id in `fontpalette.js`
- on "font-family-change" or "incre/decrement" will dispatch `font-change` event with updated values for labels || heading || axisLabels.
- default values:
- `labels.fontFamily = "Arial"` , `labels.fontSize = 12px`
- `heading.fontFamily = "Arial"` , `heading.fontSize = 16px`
- `axisLabels.fontFamily = "Arial"` , `axisLabels.fontSize = 14px`

- For e.g in `fontpalette.js` : incrementing heading font-size

- chart-heading icon



```
document.getElementById("chart-heading").addEventListener('click', function(event)
{
    that.fontChange.detail.selected = "heading";
});
```

```
document.getElementById("font-increment").addEventListener('click', function(event)
{
    var selected = that.fontChange.detail.selected;
    that.fontChange.detail[selected].fontSize += 2;
    that.fontChange.detail.changedValue = "fontSize";
    that.getPalette().dispatchEvent(that.fontChange);
    that.popDown();
});
```

- Listening to `fontChange` event trigger `onFontChange` method.

```
onFontChange(event) {
    const obj = event.detail;
    const value = obj.changedValue; // "fontSize" or "fontFamily"
    switch (obj.selected) {
        case "heading":
            pref.options.chartHeading[value] = obj.heading[value];
            break;
        case "axisLabel":
            pref.options.axisLabels[value] = obj.axisLabels[value];
            break;
        case "labels":
            // .....
            break;
```


- changing `pref/preferences` makes necessary UI changes in `ChartView` as we're watching `pref` for change and `onChange` call `this.chart.update()`.

6. Swap Fields UP and Down:



- It allows user to rearrange/order input fields hence graph.
- `selectedField` already contains the index of input fields selected
- So, swap `tabularData` as needed.

```
const index = this.selectedField;
if (index >= 1 && index <= tabularData.length-2) {
    [tabularData[index], tabularData[index-1]] =
    [tabularData[index-1], tabularData[index]];
    this.selectedField--;
}
```

- It'll swap the selected value with upper field.

7. FullScreen Mode :

```
<div id="app" :class="{ fullscreen: isFullscreen}"></div>
```

- apply fullscreen styles on `isFullscreen: true`.

```
<sugar-toolitem id="fullscreen-button" @click="isFullscreen = !isFullscreen"></sugar-toolitem>
```

- Toggle value on click.

8. Tutorial :

```
<sugar-toolitem @click="onHelp" id="help-button" title="Tutorial" class="pull-right">
</sugar-toolitem>
```

- `onHelp` will launch the tutorial.

```
var steps - [{
    title: this.l10n.stringTutoExplainTitle,
    intro: this.l10n.stringTutoExplainContent,
}, //.....
this.$refs.SugarTutorial.show(steps);
```

9. Localization :

```
<sugar-localization @localized="localized" ref="SugarL10n"></sugar-localization>
```

- Extract each text (tutorial text, title text, etc) from .po files in sugar Chart and make a locale.ini.
- Setup a l10n: { .. } with all the strings as empty.

```
localized: function () {  
    this.SugarL10n.localize(this.l10n);  
}
```

10. Journal :

- Context to be store: tabularData, pref.

```
<sugar-journal  
    ref="SugarJournal"  
    @journal-data-loaded="onJournalDataLoaded"  
></sugar-journal>
```

```
onStop: function () {  
    var context = {  
        tabularData: this.tabularData,  
        pref: this.pref,  
    };  
    this.$refs.SugarJournal.saveData(context);  
},
```

```
onJournalDataLoaded: function (data, metadata) {  
    this.tabularData: data.tabularData,  
    this.pref: data.pref,  
},
```

- With the help of above methods the data will be stored in datastore obj onStop and loaded/set when SugarJournal mounted.
- I'll add stop button as sugar-toolitem with @click = "onStop".

11. Export as image :

- Just like it's in activity Tutorial the canvas data will be exported to journal.
- `getImgData` method returns the expected data to save to journal.

```
getImgData() {  
    var mimetype = "image/png";  
    var imgData = this.chart.toBase64Image(mimetype, 1) // 1 is for high quality  
    var metadata = {  
        mimetype: mimetype,  
        title: this.activityTitle,  
        activity: "org.olpcfrance.MediaViewerActivity",  
        timestamp: new Date().getTime(),  
        creation_time: new Date().getTime(),  
        file_size: 0,  
    };  
    return { imgData, metadata };  
}
```

- We might have to update the method to give a high Quality image for MediaViewer, because at certain point even the highest quality doesn't look good on full screen.
- For that we'll resize the container and use `chart.resize()`, then `getImageURI` of canvas & finally reset chart to previous values. The resizing will not be visible to end user because of resetting it right after.
- this is the only way i found that's working after trying methods like `ctx.scale` `ctx.drawImage(..)`, they don't work as the actual chart rendering is done via `chart.js`.

12. Activity Title/ Chart Title:

- Default Title would be "Chart Activity".
- I'll attach an `EventListener` to the `activitypalette` input field.
- on change it sets title to input value-> `that.activityTitle = this.value;`

13. Sharing Activity :

- Not Completely dicussed yet ...
- Basic sharing can be easily implemented, like it's in activity tutorial.
- idea is to update tabularData & preferences on change from any shared instance see.
- idea is to update tabularData & preferences on change from any shared instance see.
- sent initial state of activity on network user list change.

```
onNetworkUserChanged(msg) {  
  if (this.SugarPresence.isHost) {  
    this.SugarPresence.sendMessage({  
      user: this.SugarPresence.getUserInfo(),  
      content: {  
        data: this.tabularData  
      },  
      .....  
    },  
    .....  
  }  
}
```

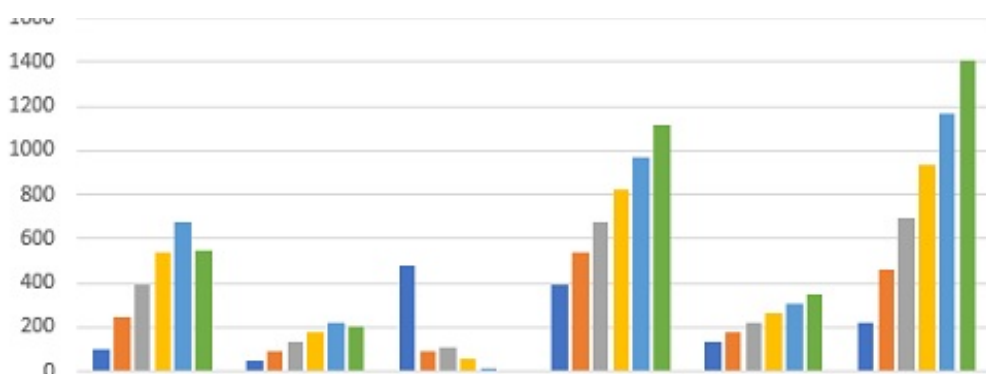
- watch for change in tabularData obj.

```
watch: {  
  tabularData {  
    handler() {  
      if (this.isUpdated) {  
        this.isUpdated = false;  
        return;  
      };  
      if (this.SugarPresence.isShared()) {  
        var message = {  
          user: this.SugarPresence.getUserInfo(),  
          content: {  
            data: this.tabularData,  
          },  
        };  
        this.SugarPresence.sendMessage(message) .....  
      }  
    }  
  }  
}
```

- on message received and update tabularData for that instance, make sure to set `isUpdated` true otherwise the change in state will again trigger sent message and forms a loop.

```
onNetworkDataReceived(msg) {
  this.isUpdated = true;
  this.tabularData = msg.content.data;
},
```

- After realizing, above method might be inefficient as we're sending whole tabular object on any change.
 - For that i've to handle all the interactions (like adding field, updating input field of particular index, etc) independently.
-
- one Idea for sharing : is to use share to compare different values at X-axis. e.g(comparing marks)
 - So onShare only host can change labels of X-axis, and all other can change the Y-axis values
 - Also we can also Sort these data for each X-label :

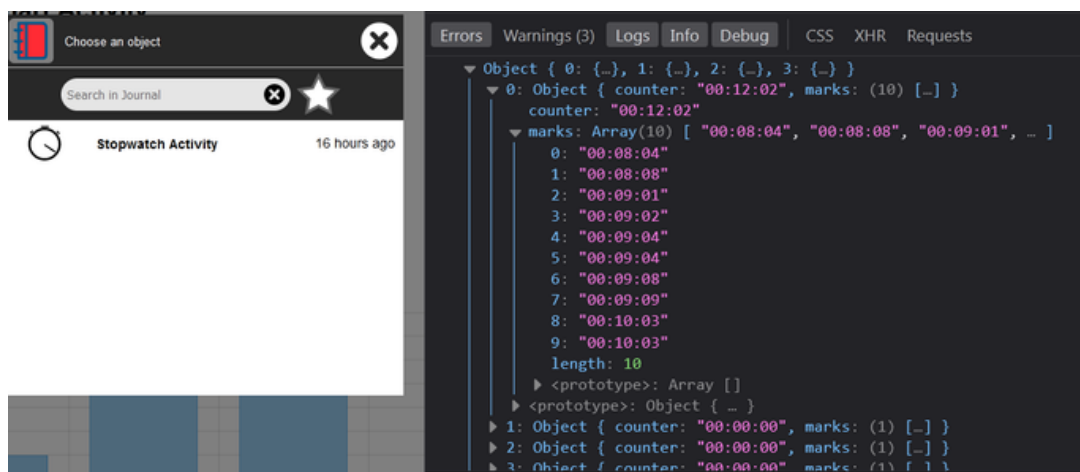


- This could be implemented using `datasets [..]` in `chart.js` and by allowing users to only change their dataset related values.

13. Read from Stopwatch & Measure :

- For Stopwatch activity : create a button which let's any one of the stop watch activity to be exported
- It has same logic of activity saving in journal ([see here](#)) rather than saving each stopwatch data we could store only the selected one
- We could also get stopwatch data directly without changing anything in Stopwatch from journal instance in journal dialog,
- but that will provide all the stopwatches data not particular one.
- The Journal instance of Stopwatch : set filters to show activity

```
var filters = [  
  { activity: "org.sugarlabs.StopwatchActivity" },  
];  
this.$refs.SugarJournal.insertFromJournal(filters).then(function (data, metadata) {  
  console.log(JSON.parse(data));  
});
```



- Use this data to populate chart.
- For Measure activity : We already have a text export to parse and use it to populate chart.
- To show it in Journal we could use "text/plain" mimetype as filter.
- { mimetype: "text/plain", activity: "org.olpcfrance.Measure"},
- These filter will not work for now due to [Issue#1351](#).
- which values to be shown in chart is yet to discuss.

14. **Extra:** nice Horizontal scrolling for large data so charts doesn't have to shrink too much.

15. Demo:

I quickly created this demo as a **proof of concept** for the methods I mentioned. Also, I have tested the aspects that are not included in the demo in smaller ways here and there.

[Demo Video ↗](#)

2. Word Puzzle Template:

- default activity obj: (to be Localized)

```
{
  "id": 7,
  "title": "WordPuzzle",
  "questions": [
    {
      "id": 1,
      "question": {
        "type": "text",
        "data": "...."
      },
      "hint": {
        "type": "text",
        "data": "....,"
      },
      "answer": "word"
    }, {...}, .....
  ],
  .....
  "type": "WordPuzzle",
  "userLanguage": "en",
  "thumbnail": "puzzle.png"
}
```

1. SETUP : files, router, & translations.

- Create Players/WordPlayerPuzzle.js, Builders/WordPlayerForm.js. Also add default obj in default_activities.json
- Add Routes for edit/play/new in containers/Router.js. e.g for /play

```
<Route
  exact
  path='/play/wordpuzzle'
  render={(props) => (
    <WORDPUZZLE
      onSharedResult={onSharedResult}
      inFullscreenMode={inFullscreenMode}
      {...props}
    />
  )}
/>
```

- Add route onPlay so when user click the play button it goes to the play route of wordpuzzle player in ExerciseList.js
- Also pass the exercise state to the route /play/wordpuzzle.

```
if (exercise.type === "WORD_PUZZLE") {
  this.props.history.push("/play/wordpuzzle", { exercise: exercise });
}
```

- Set question length and localized_type in components/Exercise.js for card title.

```
if (exercise.type === "WORD_PUZZLE") {
  length = this.props.questions.length;
  localized_type = WORD_PUZZLE;
}
```

- You have to set constant for WORD_PUZZLE containers/translation.js, the value of it determines the ID of string to be translated in different languages, present in translations/.
- Also for Localization of default activity obj we've to setup some stuff in Sugarizer.js will be discussed in Localization part.

- Players/WordPuzzlePlayer.js :

```
state: {  
  id: -1,  
  title: "",  
  questions: [],  
  submitted: false,  
  scores: [],  
  times: [],  
  currentTime: 0,  
  intervalID: -1,  
  goBackToEdit: false,  
  finish: false,  
  userLanguage: "",  
  wordList: [],  
  diagonals: true,  
}
```

- Component : <WordGrid />
- Class: PuzzleBuilder
- WordGrid is a react component which take wordList and diagonals as props and forms UI for the puzzle & also keep tracks of founded words and Score.
- PuzzleBuilder is a js class which will be used by WordGrid for creating & manipulating the grid obj which it uses to make UI.
- These class and component can be put down in WordPuzzlePlayer.js rather making another individual files.

```
<WordGrid wordList={this.wordList} diagonals={this.diagonals} />
```

```
class WordGrid extends Component {  
  constructor(props) {  
    super(props);  
  
    this.wpuzzle = new PuzzleBuilder(props.wordList, props.diagonals)  
    this.wpuzzle.setGridSize();  
    this.wpuzzle.initGrid();  
    this.wpuzzle.populateUnusedBoxes();  
  }  
  // .....
```

1. Creation of WordPuzzle grid :

- on player component mount, wordList and diagonals will be set with the values in exercise object passed through router.
- Then WordGrid will initialize wpuzzle which generates new puzzle .
- setGrid size will set the grid size such that all words fits

```
setGridSize() {  
  const offset = 2;  
  let len = this.wordList.length;  
  let list = this.wordList.slice();  
  // max length  
  let currLen = len;  
  for (let i = 0; i < len; i++) {  
    if (list[i].length > currLen) {  
      currLen = list[i].length;  
    }  
  }  
  this.gridSize = currLen + offset;  
};
```

- size = maximum of (longest word or no. of words) + offset.

- initGrid will use below described algorithms to populate this.gridArr.
- createWordGrid from wordList (Approaches):

1. Brute Force Approach (most common)

1. Select single word from wordList.
2. Select random direction & random position on WordGrid.
3. Check if word is placeable with given direction.
4. yes, Place the word, Go to step 1.
5. if not , Go to step 2.

2. !LearningApps Approach: a small Analysis [Here](#) ↗

✓ 3. Buckblog's Approach: See steps [Here](#) ↗

- uses Backtracking (appropriate method for this task, given its common use in various board games.)
 - no retry and fails like in 2nd approach.
 - places words randomly & also takes in account of already tried places unlike in 1st approach.
 - Readable & Efficient.
- populateUnusedBoxes will insert random words at empty positions in gridArr.

2. UI WordGrid :

```
let gridArr = this.wpuzzle.gridArr.slice();

{/*----- Grid array -----*/}
{gridArr.map((row, i) => {
  return (
    <div className="row" key={`row-${i}`}>
      {/*----- Row of Grid -----*/}
      {row.map((cell, i) => {
        /*----- Cell -----*/}
        return (
          <div
            className={...}
            key={`cell-${i}`}
            id={cell.id}
            onMouseDown={this.selectionStart}
            onMouseUp={this.selectionEnd}
            onMouseOver={this.mouseOver}
            onMouseOut={this.mouseOut}
          >
            {cell.letter}
          </div>
        );
      })}
    </div>
  );
})}
// .....
```

- The mouse methods will keep track of Current Selection, scores & founded words.

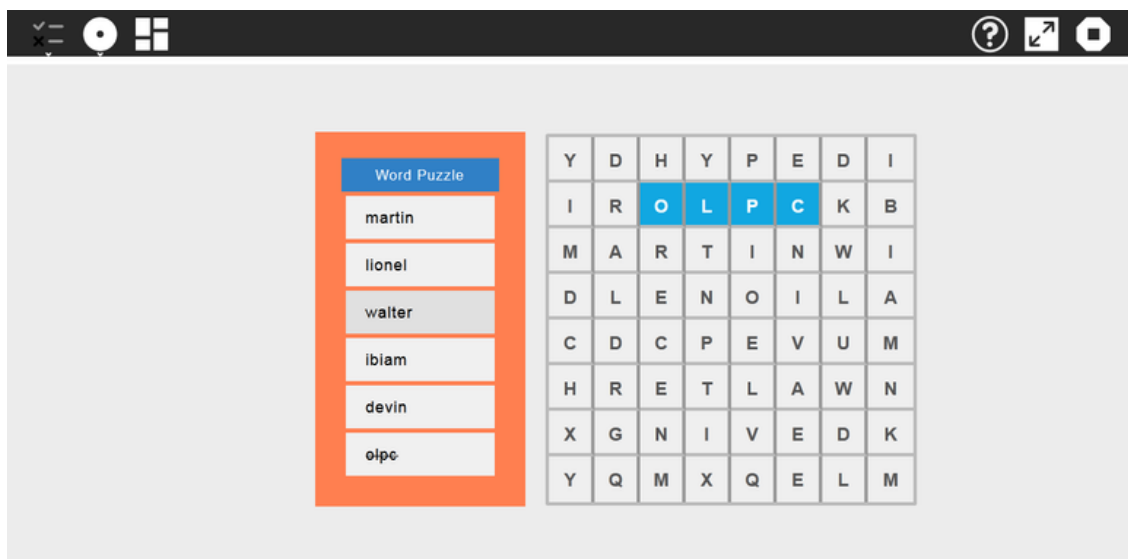
3. Show Questions with Hints

- Just like in Learning apps It'll show Questions and hints (if provided).
- To show a question and hint i'll use PlayerMultimediaJSX.

```
import { PlayerMultimediaJSX } from "../../components/MultimediaJSX";  
import { setDefaultMedia } from "../../utils";
```

```
<PlayerMultimediaJSX  
  questionType={questionType}  
  questionData={questionData}  
  speak={this.speak}  
  showMedia={showMedia}  
  willSpeak={true}  
  className="..."  
</>
```

- To Show all questions just iterate over each question to form a PlayerMultimediaJSX for each one.
- For Above to work we also have to setDefaultMedia for each question and hint, which helps to load the media required.



4. Sharing template and scores.

- Simple sharing is already handled with the help of ExerciseList.js and Sugarizer.js.
- async Evaluation : Storing activity in journal is already handled.
- We still need to determine how to display the results and scores.
- simply push /scores with the help of history object we got via withRouter

```
export default withRouter(  
  connect(MapStateToProps, {...}) //connect to redux to get state  
  (WordPuzzlePlayer)  
)
```

```
this.props.history.push("/scores", {  
  userScore: currentScore,  
  userTime: currentTime,  
  noOfQuestions: noOfQuestions,  
  exercise: exercise,  
  userAnswers: userAnswers,  
  type: "WORD_PUZZLE"  
});
```

- for async and real real time evaluation we don't show correct answers.
- We basically redirect user to new exercise (if runAll) otherwise home screen after completing exercise.
- also, Update evaluated exercise on completing. evaluation object contains all userAnswers and scores information.

```
import { updateEvaluatedExercise } from "../../store/actions/evaluation";
```

```
this.props.updateEvaluatedExercise(this.state.id, evaluation);  
if (this.props.isRunAll) {  
  this.props.history.push("/scores", {  
    next: true,  
    exercise: exercise,  
  });  
} else {  
  this.props.history.push("/");  
}
```

5. Run All exercise :

- Add the code in Score.js so nextExercise can push the correct exercise.

```
playExercise = (exercise) => {  
  //.....  
  if (exercise.type === "WORD_PUZZLE") {  
    this.props.history.push("/play/wordpuzzle", { exercise: exercise });  
  }  
};
```

6. Play and Edit existing exercise.

- Add the below in ExerciseList.js to play and edit exciting Exercise:

```
onPlay = (id) => {  
  //.....  
  if (exercise.type === "WORD_PUZZLE") {  
    this.props.history.push("/play/wordpuzzle", { exercise: exercise });  
  }  
};
```

- Do the same for onEdit method .

7. Localization : Player & Form

- For Default Object in Sugarizer.js add this code inside setDefaultExercises

```
if (  
  ..... ||  
  exercise.type === "WORD_PUZZLE"  
) {  
  for (let index = 0; index < exercise.questions.length; index++)  
    exercise.questions[index] = translateItem(exercise.questions[index]);
```

```
let translateItem = function (item) {  
  let localized = [...  
    "hint",  
    .... ]; //add hint in localized so it can detect localized string for it too  
  questions and answer are already present.
```

- Add locale strings in translations/ & also add ids of strings in translation.js, so we can use localized strings while in translateItem method.
- So now we have constants in translation.js which contains id of strings we can use it in components for e.g.

```
import {  
  SUBMIT_QUESTION,  
  NEXT_QUESTION,  
  FINISH_EXERCISE,  
  QUESTION_ERROR,  
} from "../translation";  
import { FormattedMessage } from "react-intl";
```

```
let buttonText = <FormattedMessage id={FINISH_EXERCISE} />;
```

```
question_error = (  
  <span style={{ color: "red" }}>  
    <FormattedMessage id={QUESTION_ERROR} />  
  </span>  
>;
```

- That's it for localization & Player.

- Builders/WordPuzzleForm.js :

- UI is highly inspired by the MCQ builder.
- Also the functionality of adding Title, Activity image from journal, & Questions remains same.

1. **SETUP** : Add card for WordPuzzle in html and method wordPuzzleSelected in builders/Template.js just like it's for every other template.

2. **Add Title & Image for Template** :

- Create a input form and onTitleChange method to handle title of template.

```
this.state = {
  edit: false,
  id: -1,
  title: " ",
  questions: [ ],
  isValidForm: false,
  errors: { question: false, title: false },
  currentQuestion: {
    id: 1,
    question: {
      type: "",
      data: "",
    },
    hint: {
      type: "",
      data: "",
    },
    answer: "",
  },
};
```

- to Display journal dialog popup, i'll implement showJournalChooser method which will show the appropriate media with the help of MULTIMEDIA.
- And also implement appropriate callback (e.g setQuestionSourceFromImageEditor) to set the data of that particular media chosen for that particular question/hint.

3. Add Questions and hints any media.

- For each question/hint there are two state once is media selection & other is after media selection enter value.
- If questionType is set then render QuestionJSX otherwise QuestionOptionsJSX.

```
{!questionType && (  
  <QuestionOptionsJSX  
    selectQuestionType={this.selectQuestionType}  
  />  
)}  
{questionType && (  
  <QuestionJSX  
    questionType={questionType}  
    questionData={  
      this.state.currentQuestion.question.data  
    }  
    showMedia={showMedia}  
    handleChangeQues={this.handleChangeQues}  
    speak={this.speak}  
    setImageEditorSource={  
      this.setQuestionSourceFromImageEditor  
    }  
  />  
)}  
)}
```

- Same for hint, but the hint is optional without any validation of input.
- Here the setImageEditorSource is responsible to change the state of current question and showMedia passed as prop for showing particular media type.
- For the next and previous question functionality we simply have to change the current question to the next/prev question obj.
- In case of nextQuestion store the question in questions obj.
- we don't have to checkValidation here as the next button will only be enable if everything is valid in form.
- To reset the question just change the questionType and data to empty string.

4. Test the created Exercise :

- If everything is valid the test exercise button is enabled. which let's you test the whole exercise.

```
this.props.history.push("/play/wordpuzzle", { exercise: exercise, edit: true });
```

- on clicking the test exercise it will route to /play/wordpuzzle with current exercise obj and edit true which helps the player to detect the edit mode, so it can route the user back to the /edit/wordpuzzle with exercise obj on finishing exercise.

5. Finish Exercise :

- on finish exercise just stored the current question in questions[] and then save the exercise in redux store **exercises** object.
- current question can be the last question or some intermediate question that is being edited so just check with the help of id.

```
if (currentQuestion.id <= questions.length) {  
  let Ques = {  
    id: currentQuestion.id,  
    hint: currentQuestion.hint,  
    question: currentQuestion.question,  
    answer: currentQuestion.answer,  
  };  
  questions[currentQuestion.id - 1] = Ques;  
} else {  
  questions.push({  
    id: currentQuestion.id,  
    hint: currentQuestion.hint,  
    question: currentQuestion.question,  
    answer: currentQuestion.answer,  
  });  
}
```

- Form the exercise object :
- Then save the exercise as new or in the place of the exercise being edited with the help of redux actions.

```
let exercise = {
  title: this.state.title,
  id: id,
  type: "WORD_PUZZLE",
  questions: questions,
  scores: this.state.scores,
  times: this.state.times,
  thumbnail: srcThumbnail,
  userLanguage: userLanguage,
};
```

```
import { incrementExerciseCounter } from "../../store/actions/increment_counter";
import { addNewExercise, editExercise } from "../../store/actions/exercises";
```

```
if (this.state.edit) {
  this.props.editExercise(exercise);
} else {
  this.props.addNewExercise(exercise);
  this.props.incrementExerciseCounter();
}
```

```
export default withMultimedia(require("../../media/template/wordpuzzle_image.svg"),
  withRouter(
    connect(MapStateToProps, {
      addNewExercise,
      incrementExerciseCounter,
      editExercise,
    })(WordPuzzleForm)
  )
);
```

6. Localilzation process will be same as discussed above .

This concludes the section on the implementation of the chart activity and the word puzzle template...

2023 Timeline (175 hr project)

days

May 4 - 28	■	Community Bonding Period: Finalize the features, Communicate with mentor and community members, gain a deeper understanding of code by solving issues.	25
May 29 - June 15	■	Coding officially begins: Implement Chart Activity, reproduces current features of sugar chart activity.	18
June 16 - July 9	■	Implement & test features discussed for chart activity, Base setup for word puzzle like router, defaults, state, scores, components, etc...	24
July 10 - 14	■	<u>Midterm</u> evaluations: Progress update report. (Finals week)	5
July 15 - 31	■	Word Puzzle template: implement word puzzle player with default activity obj localized & ability to share template.	17
August 1- 10	■	Implement word puzzle Form for teachers.	10
August 11 - 21	■	Implement and test, convert to evaluation functionality.	11
August 22 - 28	■	<u>Final</u> Evaluations: Final work report.	

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

- min: 30 hrs/week
- max: 50 hrs/week (if required)

How will you report progress between evaluations ?

- I will maintain a markdown document on github for it.
- I will post updates of the progress, obstacles being faced, their solutions, and also regular pull requests to sugarizer & Exerciser repo.

Discuss your post GSoC plans :

- Absolutely! The experience and exposure that I gained from just contributing is invaluable, and I am grateful for the validation it has given me. Even though i will be ineligible for next year gsoc, I understand that the main goal is to stay committed to the organization even after the GSoC program ends & i'll do it because, I am excited to see where the Sugarlabs goes in the future.

Thank you for your time & consideration ...