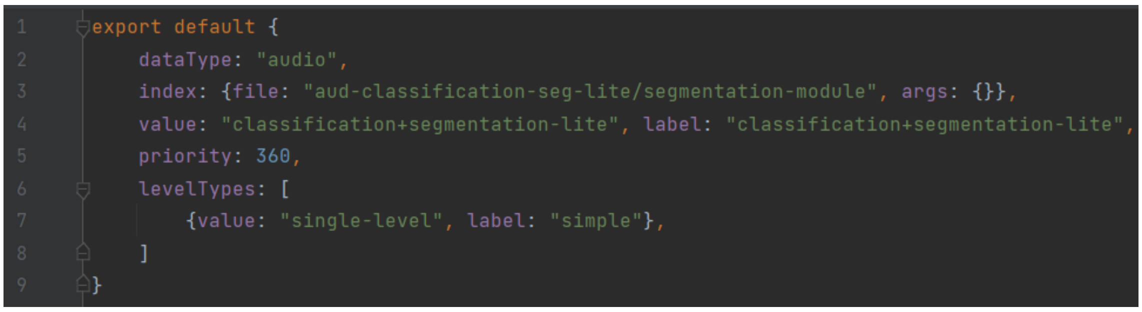
tika-work New Module Tutorial

audio classification + segmentation

This tutorial is for writing a tika-work module that has both audio classification and segmentation functionalities.

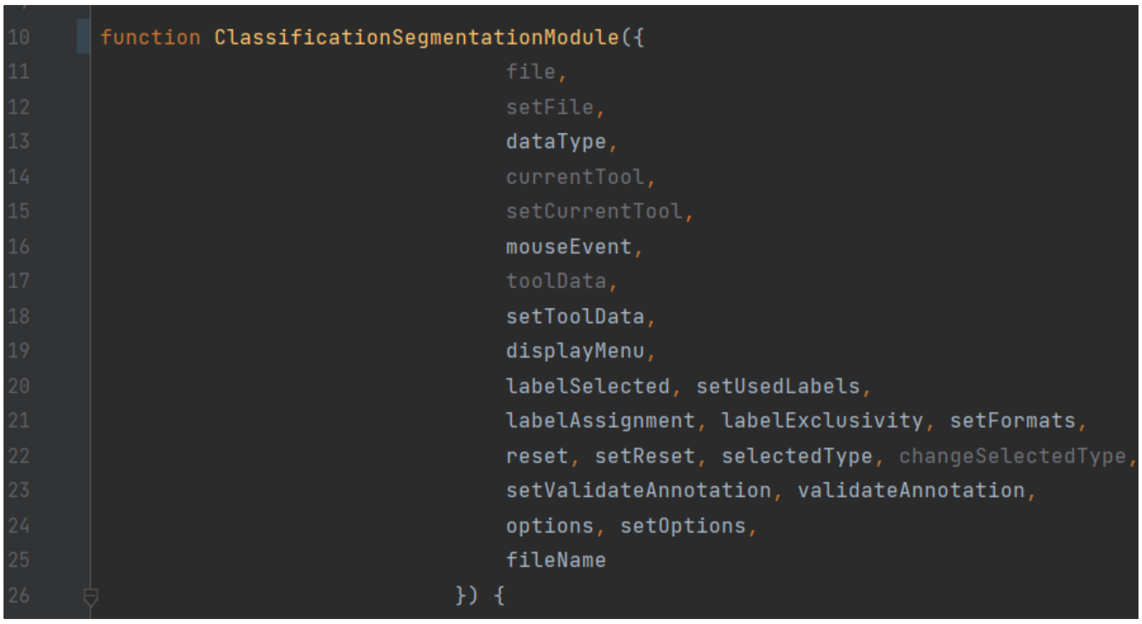
A. Basic File System

- Create the folder aud-classification-seg-lite inside the modules folder of the project.
- Add config.js file in the module folder. In this case, it will be single-type with simple mode (single-level) for labels.



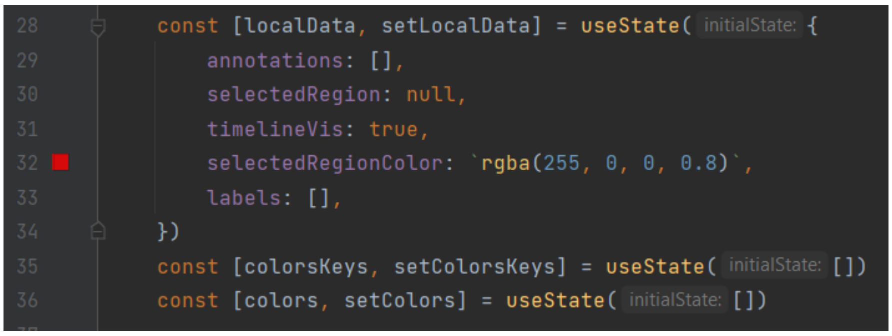
```
export default {
    dataType: "audio",
    index: {file: "aud-classification-seg-lite/segmentation-module", args: {}},
    value: "classification+segmentation-lite", label: "classification+segmentation-lite",
    priority: 360,
    levelTypes: [
        {value: "single-level", label: "simple"},
    ]
}
```

- Add the main JS file with the React Component ClassificationSegmentationModule and use the parameters that will be passed by the main code when the module is loaded.



```
function ClassificationSegmentationModule({
                                  file,
                                  setFile,
                                  dataType,
                                  currentTool,
                                  setCurrentTool,
                                  mouseEvent,
                                  toolData,
                                  setToolData,
                                  displayMenu,
                                  labelSelected, setUsedLabels,
                                  labelAssignment, labelExclusivity, setFormats,
                                  reset, setReset, selectedType, changeSelectedType,
                                  setValidateAnnotation, validateAnnotation,
                                  options, setOptions,
                                  fileName
                              }) {
```

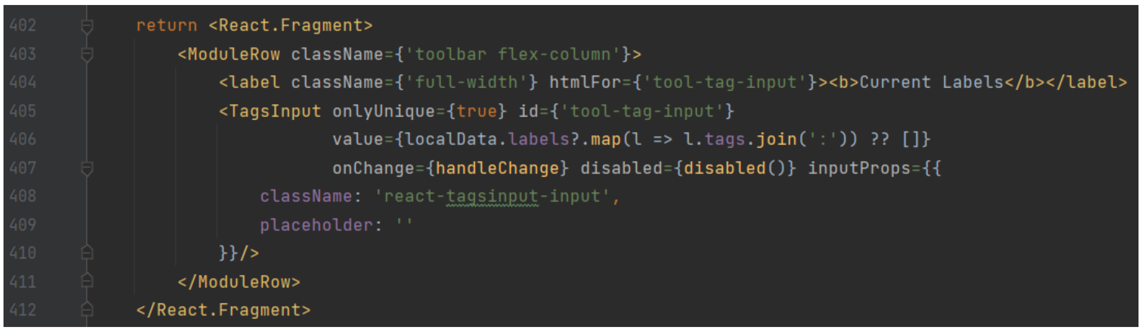
- Use custom variables to set the data state. LocalData for annotations, selected region, visual configurations and classification labels. Also, the list of colors to use on the regions.



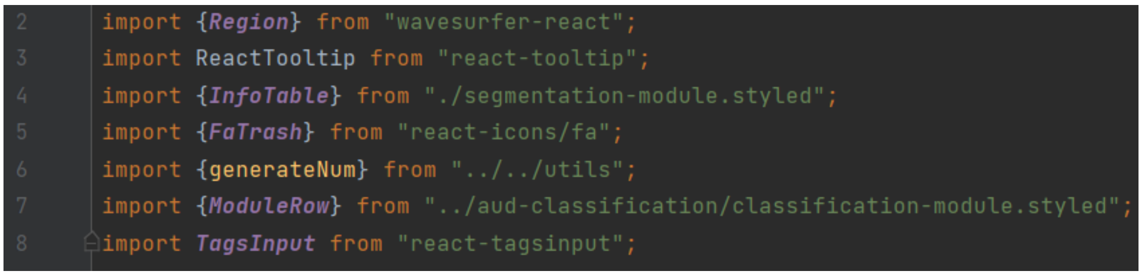
```
const [localData, setLocalData] = useState( initialState: {
    annotations: [],
    selectedRegion: null,
    timelineVis: true,
    selectedRegionColor: `rgba(255, 0, 0, 0.8)`,
    labels: [],
})
const [colorsKeys, setColorsKeys] = useState( initialState: [])
const [colors, setColors] = useState( initialState: [])
```

B. React Html components

- Return React Html to show module custom controls for the user interface. You can use external, internal, or styled-components. It has one toolbar with the tags input control.

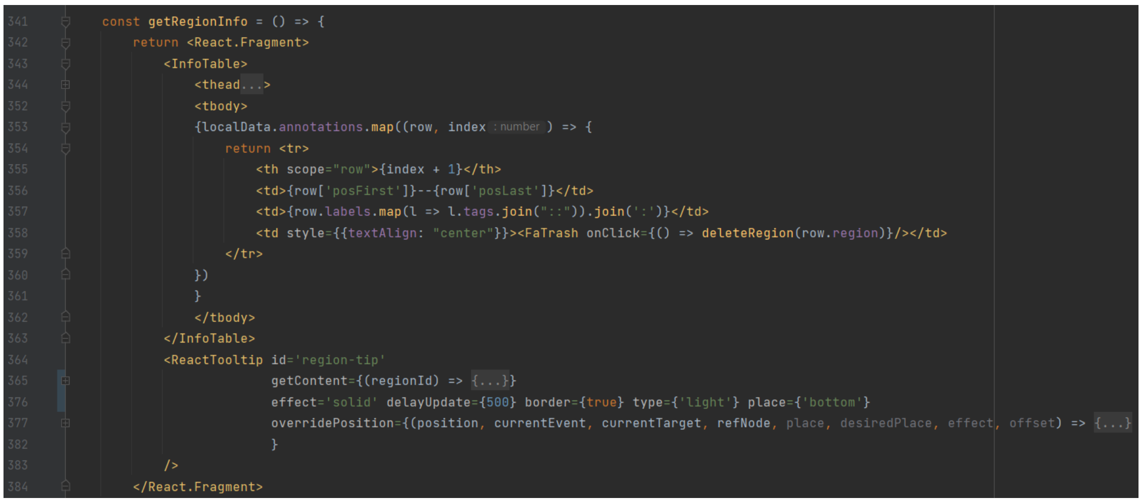


```
return <React.Fragment>
    <ModuleRow className={'toolbar flex-column'}>
        <label className={'full-width'} htmlFor={'tool-tag-input'}><b>Current Labels</b></label>
        <TagsInput onlyUnique={true} id={'tool-tag-input'}
                   value={localData.labels?.map(l => l.tags.join(':')) ?? []}
                   onChange={handleChange} disabled={disabled()} inputProps={{
            className: 'react-tagsinput-input',
            placeholder: ''
        }}/>
    </ModuleRow>
</React.Fragment>
```



```
import {Region} from "wavesurfer-react";
import ReactTooltip from "react-tooltip";
import {InfoTable} from "./segmentation-module.styled";
import {FaTrash} from "react-icons/fa";
import {generateNum} from "../../utils";
import {ModuleRow} from "../aud-classification/classification-module.styled";
import TagsInput from "react-tagsinput";
```

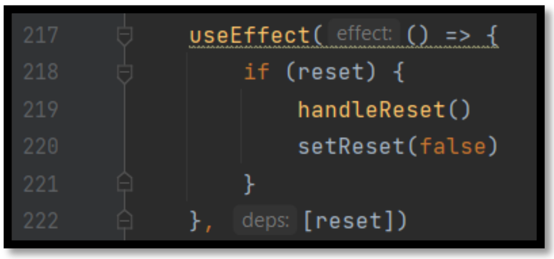
- The audio segmentation viewer allows extra React Html to show.



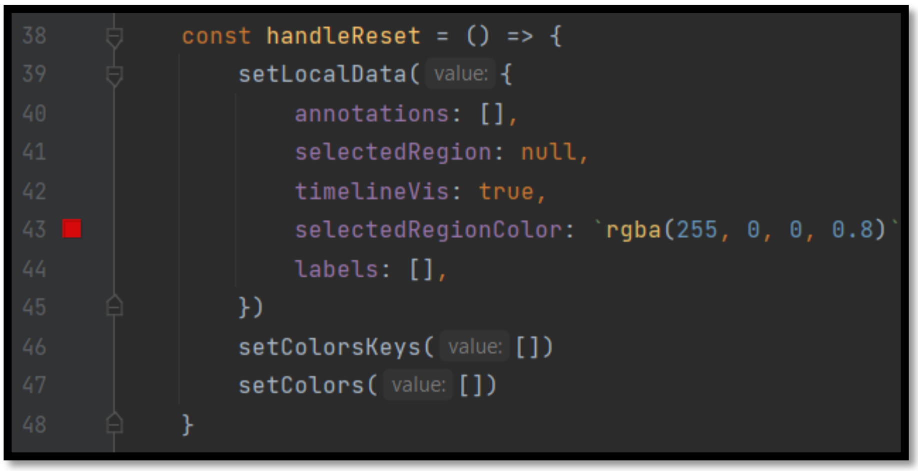
```jsx
const getRegionInfo = () => {
    return <React.Fragment>
        <InfoTable>
            <thead...>
            <tbody>
            {localData.annotations.map((row, index) => {
                return <tr>
                    <th scope="row">{index + 1}</th>
                    <td>{row['posFirst']}--{row['posLast']}</td>
                    <td>{row.labels.map(l => l.tags.join("::")).join(':')}</td>
                    <td style={{textAlign: "center"}}><FaTrash onClick={() => deleteRegion(row.region)}/></td>
                </tr>
            })
            }
            </tbody>
        </InfoTable>
        <ReactTooltip id='region-tip'
                      getContent={(regionId) => {...}}
                      effect='solid' delayUpdate={500} border={true} type={'light'} place={'bottom'}
                      overridePosition={(position, currentEvent, currentTarget, refNode, place, desiredPlace, effect, offset) => {...}
                      }
        />
    </React.Fragment>
```

C. Handling events from the platform

- On the other hand, it is necessary to keep the module listening for external state changes to perform the indicated actions or respond to events. E.g., when changing the reset option set local data to the initial state.

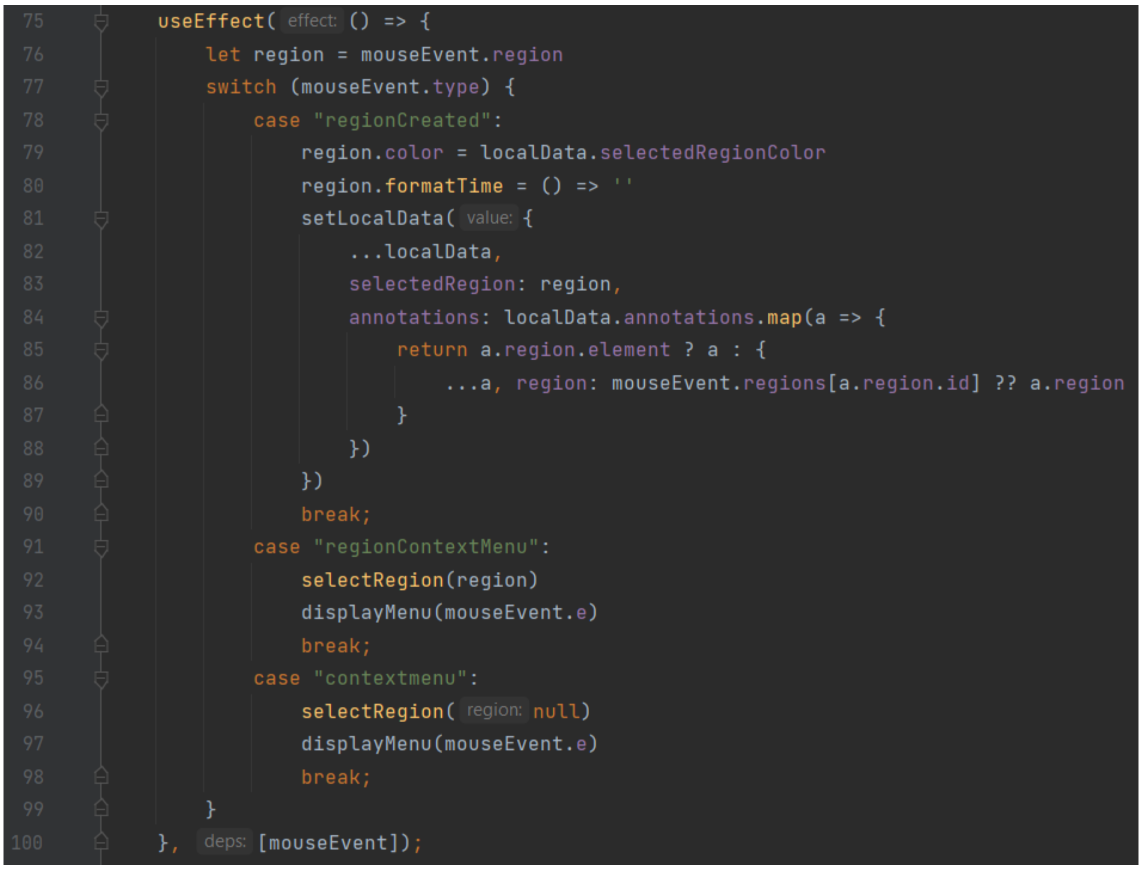


```jsx
useEffect(() => {
    if (reset) {
        handleReset()
        setReset(false)
    }
}, [reset])
```



```jsx
const handleReset = () => {
    setLocalData({
        annotations: [],
        selectedRegion: null,
        timelineVis: true,
        selectedRegionColor: `rgba(255, 0, 0, 0.8)`,
        labels: [],
    })
    setColorsKeys([])
    setColors([])
}
```

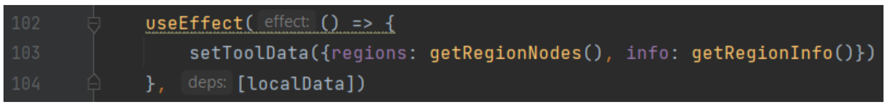
- React to each event depending on the action on the specific viewer. When a region is created, it updates the selected region and the regions in the list of existing annotations. In cases of contextual menu events, the corresponding region is selected and the menu with the options is displayed.



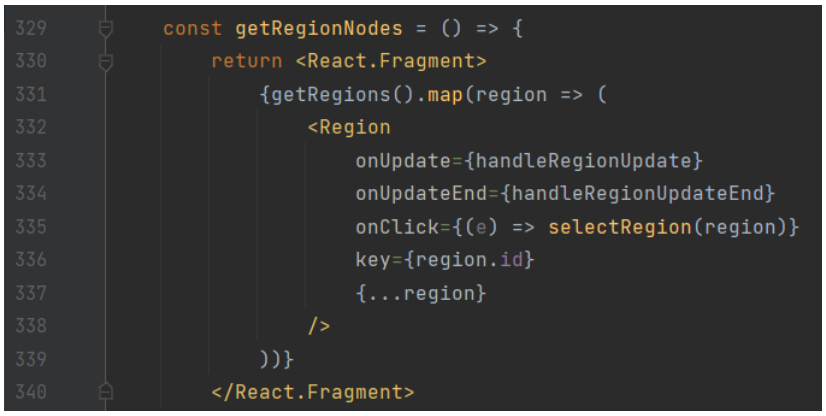
```
useEffect(() => {
    let region = mouseEvent.region
    switch (mouseEvent.type) {
        case "regionCreated":
            region.color = localData.selectedRegionColor
            region.formatTime = () => ''
            setLocalData({
                ...localData,
                selectedRegion: region,
                annotations: localData.annotations.map(a => {
                    return a.region.element ? a : {
                        ...a, region: mouseEvent.regions[a.region.id] ?? a.region
                    }
                })
            })
            break;
        case "regionContextMenu":
            selectRegion(region)
            displayMenu(mouseEvent.e)
            break;
        case "contextmenu":
            selectRegion(null)
            displayMenu(mouseEvent.e)
            break;
    }
}, [mouseEvent]);
```

D. Graphic representation

- In the case of segmentation modules, it is possible to draw regions over an audio wave sending a list of region objects.



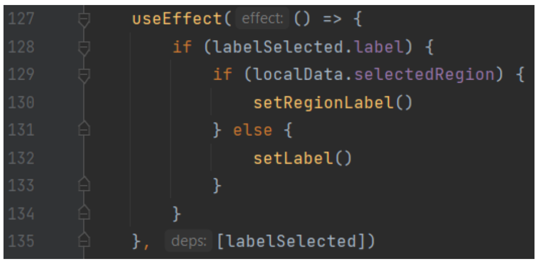
```
useEffect(() => {
    setToolData({regions: getRegionNodes(), info: getRegionInfo()})
}, [localData])
```



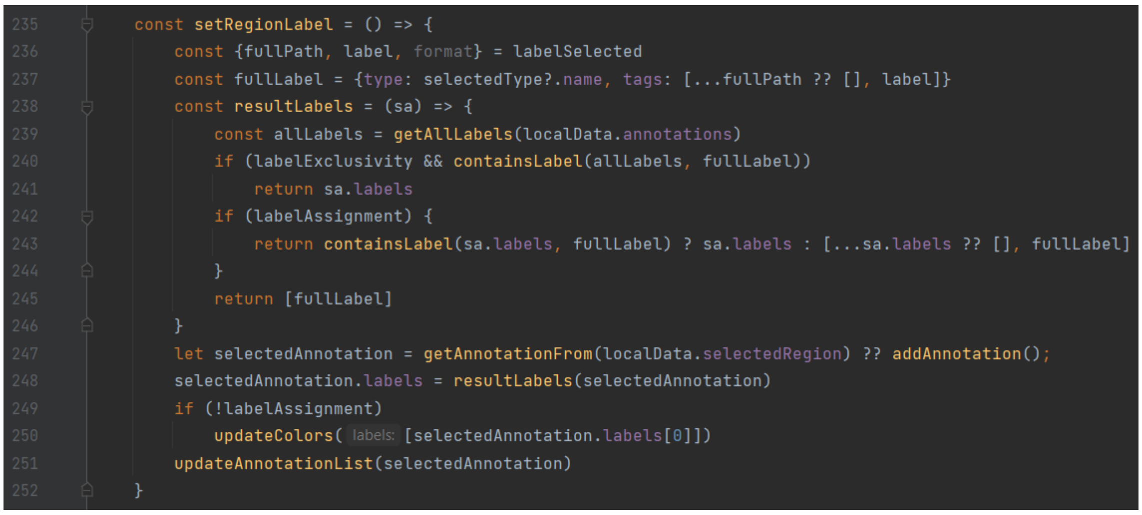
```
const getRegionNodes = () => {
    return <React.Fragment>
        {getRegions().map(region => (
            <Region
                onUpdate={handleRegionUpdate}
                onUpdateEnd={handleRegionUpdateEnd}
                onClick={(e) => selectRegion(region)}
                key={region.id}
                {...region}
            />
        ))}
    </React.Fragment>
```

E. Annotations

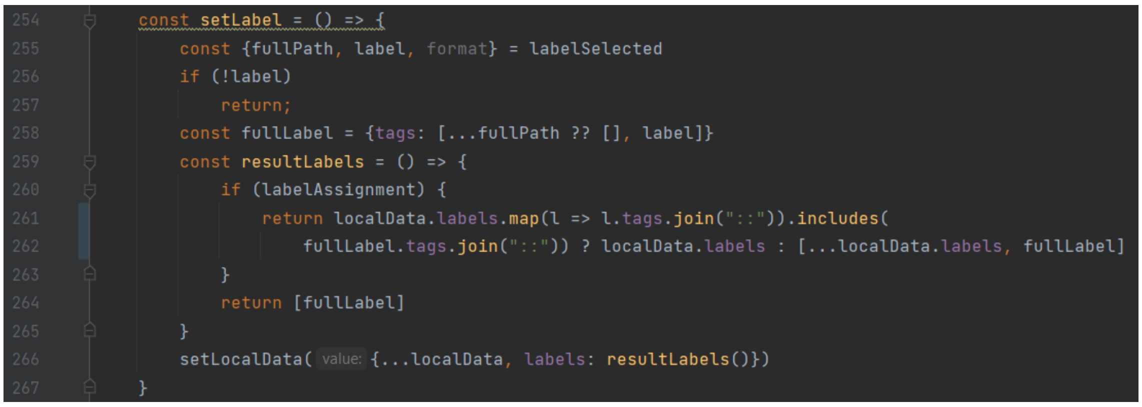
- When a label is selected on the context menu the variable labelSelected will change the value with the data of the label. You must consult labelExclusivity and labelAssignment to respect the logic of the system and assign the label to the selected region.



```
useEffect( effect: () => {
    if (labelSelected.label) {
        if (localData.selectedRegion) {
            setRegionLabel()
        } else {
            setLabel()
        }
    }
}, deps: [labelSelected])
```

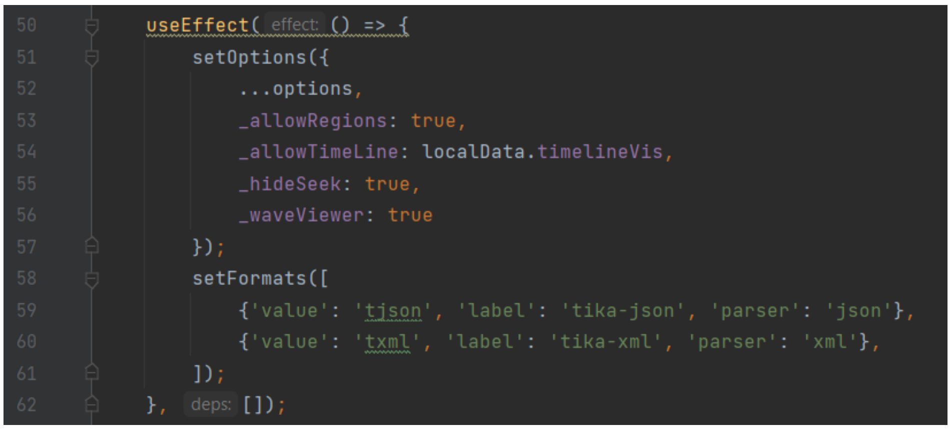


```
const setRegionLabel = () => {
    const {fullPath, label, format} = labelSelected
    const fullLabel = {type: selectedType?.name, tags: [...fullPath ?? [], label]}
    const resultLabels = (sa) => {
        const allLabels = getAllLabels(localData.annotations)
        if (labelExclusivity && containsLabel(allLabels, fullLabel))
            return sa.labels
        if (labelAssignment) {
            return containsLabel(sa.labels, fullLabel) ? sa.labels : [...sa.labels ?? [], fullLabel]
        }
        return [fullLabel]
    }
    let selectedAnnotation = getAnnotationFrom(localData.selectedRegion) ?? addAnnotation();
    selectedAnnotation.labels = resultLabels(selectedAnnotation)
    if (!labelAssignment)
        updateColors( labels: [selectedAnnotation.labels[0]])
    updateAnnotationList(selectedAnnotation)
}
```



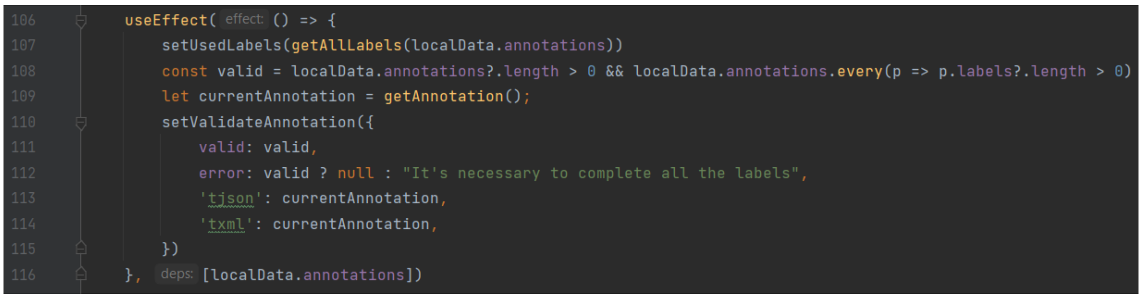
```
const setLabel = () => {
    const {fullPath, label, format} = labelSelected
    if (!label)
        return;
    const fullLabel = {tags: [...fullPath ?? [], label]}
    const resultLabels = () => {
        if (labelAssignment) {
            return localData.labels.map(l => l.tags.join("::")).includes(
                fullLabel.tags.join("::")) ? localData.labels : [...localData.labels, fullLabel]
        }
        return [fullLabel]
    }
    setLocalData( value: {...localData, labels: resultLabels()})
}
```

- The module must specify the formats that it allows to export and declare what type of data corresponds to each one, XML or JSON. Other display options can also be set from the start too.



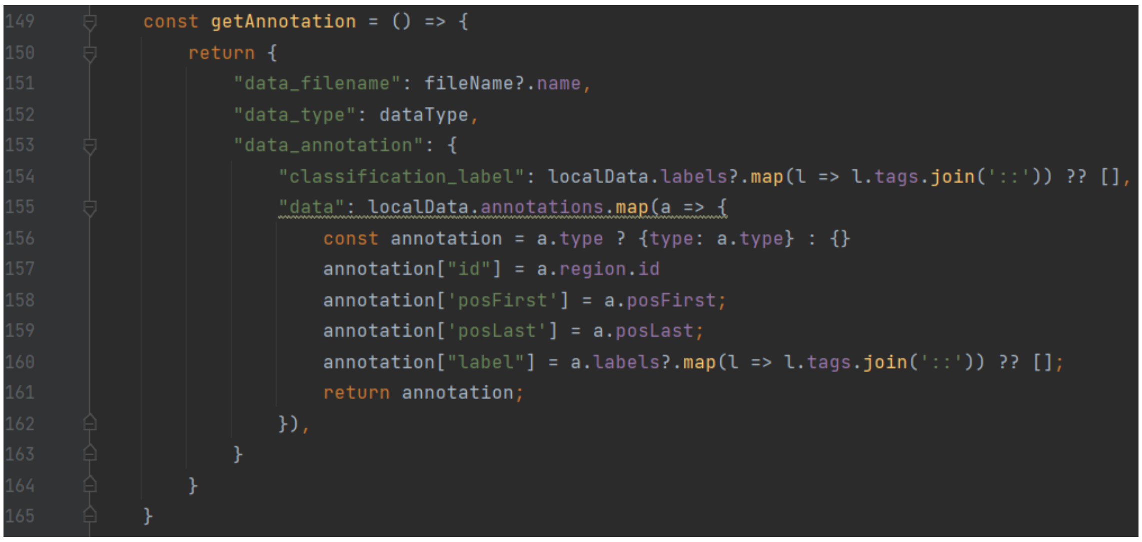
```
    useEffect(() => {
        setOptions({
            ...options,
            _allowRegions: true,
            _allowTimeLine: localData.timelineVis,
            _hideSeek: true,
            _waveViewer: true
        });
        setFormats([
            {'value': 'tjson', 'label': 'tika-json', 'parser': 'json'},
            {'value': 'txml', 'label': 'tika-xml', 'parser': 'xml'},
        ]);
    }, []);
```

- The module information must be kept updated after each change, indicating if it is valid for export, the data for each format or the existing error.



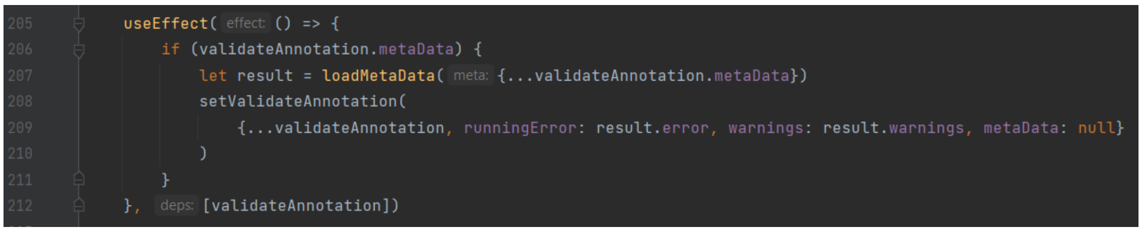
```
    useEffect(() => {
        setUsedLabels(getAllLabels(localData.annotations))
        const valid = localData.annotations?.length > 0 && localData.annotations.every(p => p.labels?.length > 0)
        let currentAnnotation = getAnnotation();
        setValidateAnnotation({
            valid: valid,
            error: valid ? null : "It's necessary to complete all the labels",
            'tjson': currentAnnotation,
            'txml': currentAnnotation,
        })
    }, [localData.annotations])
```

- The data to be exported depends in a general sense on the specific format being exported and the type of data. The most specific data of the annotation are obtained from the lists of annotations stored in the local data and the assigned labels.



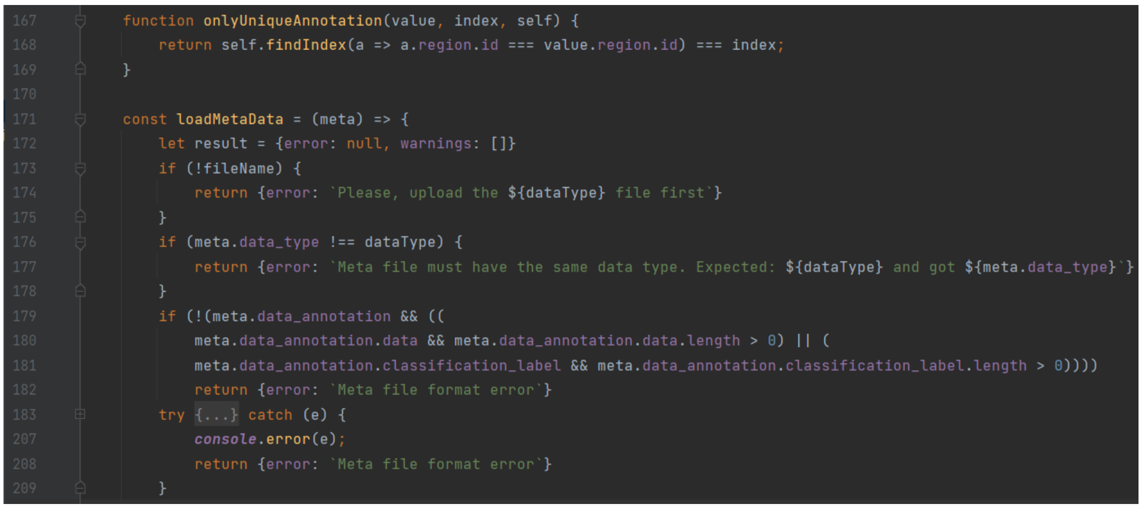
```
    const getAnnotation = () => {
        return {
            "data_filename": fileName?.name,
            "data_type": dataType,
            "data_annotation": {
                "classification_label": localData.labels?.map(l => l.tags.join('::')) ?? [],
                "data": localData.annotations.map(a => {
                    const annotation = a.type ? {type: a.type} : {}
                    annotation["id"] = a.region.id
                    annotation['posFirst'] = a.posFirst;
                    annotation['posLast'] = a.posLast;
                    annotation["label"] = a.labels?.map(l => l.tags.join('::')) ?? [];
                    return annotation;
                }),
            }
        }
    }
```

- If a metadata file has been imported, the change is reported through the variable validateAnnotation.metaData which should be processed and reset to null.



```
useEffect( effect: () => {
    if (validateAnnotation.metaData) {
        let result = loadMetaData( meta: {...validateAnnotation.metaData})
        setValidateAnnotation(
            {...validateAnnotation, runningError: result.error, warnings: result.warnings, metaData: null}
        )
    }
}, deps: [validateAnnotation])
```

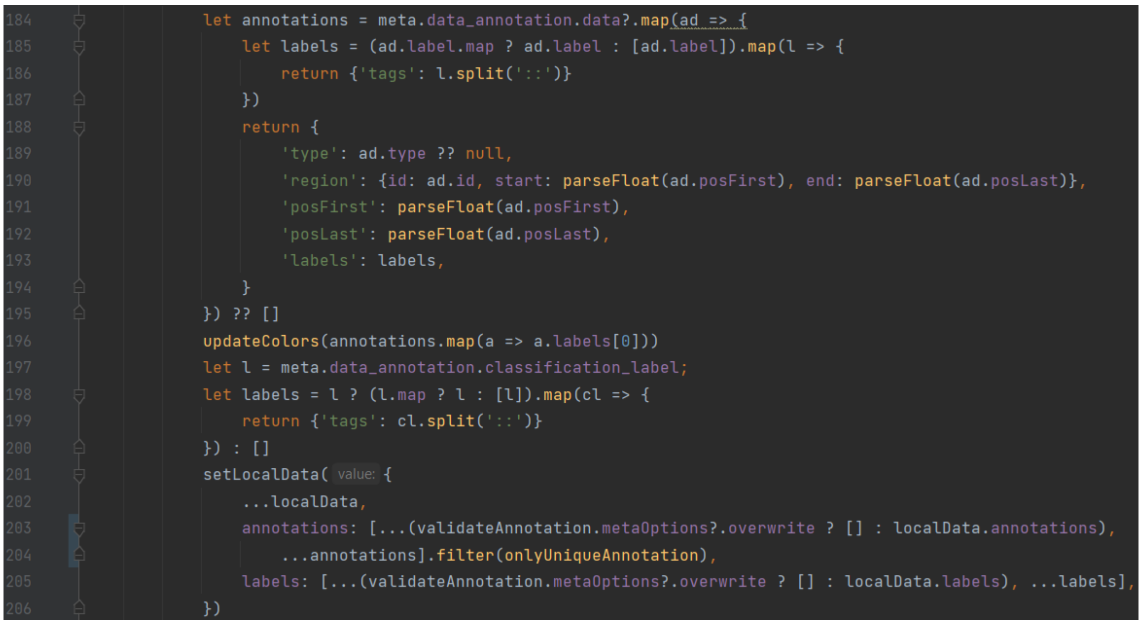
- During the import process it is possible to make the pertinent verifications and return the error or the necessary warnings.



```
function onlyUniqueAnnotation(value, index, self) {
    return self.findIndex(a => a.region.id === value.region.id) === index;
}

const loadMetaData = (meta) => {
    let result = {error: null, warnings: []}
    if (!fileName) {
        return {error: `Please, upload the ${dataType} file first`}
    }
    if (meta.data_type !== dataType) {
        return {error: `Meta file must have the same data type. Expected: ${dataType} and got ${meta.data_type}`}
    }
    if (!(meta.data_annotation && ((
        meta.data_annotation.data && meta.data_annotation.data.length > 0) || (
        meta.data_annotation.classification_label && meta.data_annotation.classification_label.length > 0))))
        return {error: `Meta file format error`}
    try {...} catch (e) {
        console.error(e);
        return {error: `Meta file format error`}
    }
```

- After the verifications are done, the objects must be rebuilt according to the module. Verifying the variable validationAnnotation.metaOptions.overwrite allows knowing the mod to proceed with the existing data in the local data.



```
        let annotations = meta.data_annotation.data?.map(ad => {
            let labels = (ad.label.map ? ad.label : [ad.label]).map(l => {
                return {'tags': l.split('::')}
            })
            return {
                'type': ad.type ?? null,
                'region': {id: ad.id, start: parseFloat(ad.posFirst), end: parseFloat(ad.posLast)},
                'posFirst': parseFloat(ad.posFirst),
                'posLast': parseFloat(ad.posLast),
                'labels': labels,
            }
        }) ?? []
        updateColors(annotations.map(a => a.labels[0]))
        let l = meta.data_annotation.classification_label;
        let labels = l ? (l.map ? l : [l]).map(cl => {
            return {'tags': cl.split('::')}
        }) : []
        setLocalData( value: {
            ...localData,
            annotations: [...(validateAnnotation.metaOptions?.overwrite ? [] : localData.annotations),
                ...annotations].filter(onlyUniqueAnnotation),
            labels: [...(validateAnnotation.metaOptions?.overwrite ? [] : localData.labels), ...labels],
        })
```

*** This is only a variant for the implementation, the platform provides freedoms for the design, the objects and events that want to be represented by the SVG object*