Important!!

This document contains a few guidelines for the skynote code. It's a work in progress, so if you notice something outdated or missing, feel free to make any corrections or additions.
The original file I used is HERE :)

Clarification!!

This is not as relevant, but there is usually a mix-up when it comes to the word “repetitions”. It is usually used to describe the amount of times a user is meant to play a specific part of the score (defined by the repeat marks). But it is also used to describe the amount of times a user might want to play the whole score. At the moment we don’t have a consensus on how to distinguish between these two cases, so it is possible that the code might be confusing in this specific area. In order to make things simple, here I will use “repetitions” for times a score is played, and “sub-repetitions” for parts of the score that need to be played twice (this difference does not exist in the code right now), the way it works currently is as follows:

- **Recording mode** only allows the user to play once the score(then they have to save it or delete it before creating a new one). This means that only 1 repetition is allowed, no matter the amount of sub-repetitions.
- **Practice mode** allows the user to play as much as they want, meaning that they can have as many repetitions as they want (it is advisable to look into that, and maybe limit the amount of repetitions, but that data doesn’t go to the DB, so is not very relevant right now). The amount of sub-repetitions is also limitless.

In terms of code, both repetitions and sub-repetitions are handled equally. It is easier to explain it with an example, so I’ll use “Surprise” (level 2 ,Pieces V):
Surprise has 3 sub-repetitions, the first 2 have a different ending, and the last one doesn’t. If a recording for the piece is checked, it can be seen that data for sub-repetition 1 goes from measures 1 to 8, sub-repetition 2 goes from 1 to 17, and sub-repetition 3 goes from 10 to 17. The reason behind this is that we don’t want data lines to overlap, so we store every sub-repetition separately.

Tips & tricks

A few tips and tricks I find useful.

- As explained by Amaia, using % in the VScode search bar allows to search specific across all the files in the project, very useful to find specific functions, exports or literally anything in the project.
- Using “//#region” before a chunk of code, and “#endregion” after the chunk, allows you to collapse all the code in that specific region, and it also creates tags in the “Minimap”(or Code Outline/Document Outline). Very useful when going through files with a lot of lines of code.

2. Main .js files

Here is a list of the main .js files, in the order the user would see them when using skynote for the first time.

Landing.js

When the user arrives at the site for the first time, they see “Landing.js”. Here they can navigate to “Demos.js”, “OurTeam.js”, “Research.js”, Interest Form(redirects to a Google Form), and “Register.js”.

Register.js

When the user clicks the submit button, “Register.js” makes use of appContext.js

```
const { user, isLoading, showAlert, displayAlert, setupUser } =
useAppContext();
```

Here, “appContext.js” is used to log in the user, and store some relevant information locally. This is done through “addUserToLocalStorage” and “getAllScoreData2”. I believe that the original intention was to use JWT tokens to store all this information, but this is currently not implemented.

Once the user is logged in, “Stats.js” is loaded. (This file is explained in the document that Amaia wrote before leaving, and should be merged into this one. It contains info about the Dashboard, All lessons, My recordings and Assignments menus)

Other than that, there are 3 extra menus:

- Profile: “Profile.js” is a simple file that allows the user to change some of the user data from the database. It has one useEffect() to get the user data from the database, and another one to register every change made in the form.
- Sound visualization: “TimbreVisualization.js” is called here. The user can use their mic to analyse some aspects of their playing like the pitch or the dynamic stability.
- API testing: “apitesting.js” contains a few functions for trying some of the database calls.
    - toggletimer: allows to activate a timer. The idea behind this is to try and record the amount of time a user is actually using skynote.
    - Create task: this one allows for the creation of tasks, in order to help with testing. It does that by creating a popup window (PopUpWindowAssignments.js) At the moment the list of students and the teacher Id are fixed
        - The teacher part if the site is not done yet, so we set a fixed teacherId just for testing purposes.
        - The list of students should be retrieved from the database, but we haven’t decided on how we should store the relationship between student and teacher. We could assign a teacherId to every student and then search all the students that match a specific teacherId, or we could add a “contacts” field to every user, and that way we could ask for the info of one specific database object, rather than going over all of them. This also

allows for future features like adding friends, but we are currently not focused on that yet.

- Grade task: this hasn't been coded yet, so right now it only shows an empty popup window ("PopUpWindowGrading.js"). The focus is not on this at the moment, so it's probably not worth investing time into this, and code it once the teacher side is created.
- postMessage: This one posts a prewritten message in the chat from the current user to our default teacherId (in this case '5d34c59c098c00453a233bf3')
- getMessages: This one gets the latest messages in descending date order. The limit of messages can be chosen, but it's currently fixed to 12.

3. DB calls

In order to create new database calls, the file "flow" is file, then methods, then routes, then controller and finally model. An example of this would be:

- The file "ListAllRecordings.js" uses a function called "getAllRecData" to retrieve all of the recordings from a given user (studentId).
- The "getAllRecData" function is in "studentRecordingMethods.js". It is an async function that makes an axios.get request to the URL endpoint "/api/v1/recordings/getAllRecData".
- The "recordings/getAllRecData" route is made in "recordingRoutes.js". When a GET request is made to this route, it is handled by the "getAllRecData" function, which is imported from "recordingController.js"
- In "recordingController.js", the function "getAllRecData" takes the mongoose schema from "StudentRecordings.js" (in the models folder), as "student_recordings" and makes all the necessary database calls.

There are models for every DB collection, and the same goes for controllers and routes:

Route	Controller	Model	Collection
assignmentRoutes.js	assignmentController.js	Assignments.js	assignments
messageRoutes.js	messageController.js	Message.js	messages
scoreRoutes.js	scoreController.js	xmlScoreModel.js	scores
recordingRoutes.js	recordingController.js	StudentRecordings.js	student_recordings
authRoutes.js	authController.js	User.js	users

4. Styling

Styling is done though the CSS module files with the same name as the javascript files. For example, the styling for 'MainMenu.js' is done in 'MainMenu.module.css'. We changed to this way of styling not so long ago, so there is a chance that some of the older files don't have a '.module.css' file. If that is the case, a new '.module.css' file should be eventually created.
Other than that, some other rules are being applied that come from the "normalize.css" package, from Bootstrap and from the "index.css" file.
I've found a few inconsistencies in the form of redundant rules, or rules that are applied at the wrong time (for example, rules from ListRecordings.module.css applied to the assignments page). A few things to keep in mind:

- The .module.css files should not make changes to tags like <body>, as this causes the css to affect the whole page rather than specific areas. This should be done in index.css. The .module.css files should only make changes by className.

In order to keep consistency, this is a set of colours we've been using, if the colour you're looking for is not here, feel free to add it to the list, but make sure you're not assigning a new colour to something that already exists (there is also a list of colours in "index.css", but I believe some of the newer .module.css files don't necessarily take that list into account):

- Back button : #9E9E9E
- Backbutton hover: #717171
- Borders: #CADDFB
- Box-shadow: #CADDFB
- Containers (aka background colour): #FFFFFF
- Fonts: #000000
- Icons: #939393
- Icon button hover: #D2E8E1(this one needs to be reviewed)
- Star complete: #E8D73F
- Star incomplete: #909090

5. ProgressPlayFile.js

This file, and OpenSheetMusicDisplay.js are the biggest in terms of lines of code, and thus they deserve a special section, since they also happen to be the most relevant ones.
ProgressPlayFile.js is in charge of managing everything that happens when the user is looking at the score, some of the functions are delegated to other files (ControlBar.js, audioStreamer.js…), but ProgressPlayFile.js is the base for everything. I'll try to explain a bit of what's going on, following the way it's written in the code.
The code is structured by variables, functions, hooks, and the return. There are a few comments in the code itself, so I'll avoid getting too detailed if it's not needed.

The code starts with around 60 lines of variable declarations. Mostly flags for stuff to activate/deactivate, plus a few ones that are used to store data (pitch line, note color…).

After that, there are 9 functions (at the time of this being written) in charge of coordinating basically everything between OpenSheetMusicDisplay.js, audioStreamer.js, the control bars, the metronome countdown and the popup windows.

- "handleReceiveRepetitionInfo", "onResetDone", "handleFinishedCursorOSMDCallback" and "handleGetJsonCallback" are sent to OpenSheetMusicDisplay.js, basically to get information from the score from osmd.
- "handlePitchCallback", and "aCb" are sent to audioStreamer.js in order to get information about the different audio features.
- The rest of them don't require further explanation than then one in the code.

Next, there are a few useEffect() hooks that are used to keep track of every change that might happen in the page. All of them are explained in the code.

Finally, the return is explained in the summary that Amaia left.

6. OpenSheetMusicDisplay.js

This is a very condensed summary of the file, but essentially, OpenSheetMusicDisplay.js (OSMD from now on), gets information about the pitch, then translates it to an offset in the staff (B4 being the middle of the staff), and uses that offset to draw the pitch line. Additionally, since it's the only file that makes use of the "opensheetmusicdisplay" library, it is also in charge of dealing with all the score properties like playing the score, adjusting the volume and the zoom, etc…

Right now the way the pitch line is drawn is by following the cursor. Every time the cursor moves to a new note, the line jumps to the new position and starts drawing at a constant speed based on the pitch input (this means that every note has its own line).

As the cursor follows the score, the note under the cursor is extracted (check //#region EXTRACT NOTE POSITION in the code), and based on the *pitch confidence*[1], it assigns either green or red colors to the note being played. Finally, there are a few arrays being pushed with every note, that are in charge of storing all the information regarding note colors, their IDs and the pitch line attached to them. We need the note IDs to associate them with the SVG elements.

This represents a very clear issue because since the cursor from osmd moves in a discrete motion (meaning it jumps from one note to another), it is very difficult to create a nice continuous pitch line.

Another option would be to not follow the cursor, but for that we would need to know the speed at which the line should be drawn. This problem is also not trivial, because every measure requires a different speed based on the length of the measure in pixels (which also depends on the zoom a user might set for the score, and the

[1] *In here with pitch confidence I'm not talking about the one that MEYDA returns, we check the pitch for the duration of the note, and if the measured confidence is .5 or more, we "paint" that note green*

BPM). Doing this in real time is probably not the best option, so we would need to read every .xml file when a user loads it, check every measure length and then apply a “speed factor” for every measure.

Following this idea, another way would be to extract the info about the measures everytime we upload a new .xml to the database, and then storing that information as “score properties” or something like that. That way every time a user loads a score we can also load the “properties” of said score and keep adjusting the speed of the line according to those.

This is just a theory of mine, and I’m not sure how doable this is, mainly because I believe osmd doesn’t give a lot of information about measure length. If we decide to go down the preprocessing rabbit hole, we can also create new properties with more information about measure position, meaning that we might be able to completely separate the pitch line drawing from osmd. This is also a nice option because it would mean that the whole project is not as heavily dependent on osmd.

7. Small bugs/things I have encountered

- In OpenSheetMusicDisplay.js
    i. This file deals with everything related to OSMD. I’m not sure how doable it might be, but I think it might be interesting to look into breaking it into smaller files, for example one file that deals with everything related to drawing the pitchline and extracting the positions of the notes, another file that deals with the data, and another one that deals with stuff like volume/zoom changes, etc…
    ii. If the user doesn’t record a full score, they will always get zero stars. This was done because at that time it was the simplest thing, but this can be improved so they get a maximum of one star if they have completed the first third of the score, a max of two stars if they complete two thirds of the score…
    iii. Just before the “return”, there is a const called “lineChartStyle”. Styling should not be done there.