Speechviz Design Implementation

Features:

- Displaying speaker separation and VAD/Non-VAD (controlled in scripts/process_audio.py)
- Displaying extracted information from VRS files recorded by project ARIA glasses (extracted with scripts/extract-vrs-data.py)
    - Pose/orientation of the glasses (controlled in scripts/create_poses.py)
    - Face Clustering (controlled by scripts/encode_and_cluster.py)
- Displaying speech to text (controlled in scripts/transcribe.py)
- Labeling new speakers
- Adding custom segments
- Moving/renaming segments
- Associating clustered faces with Speakers
- Changing colors shown on display
- Estimates SNR (signal to noise ratio) and ranks speakers based upon snrs and duration
    - What it thinks is the primary speaker is put in pink and is number 1.
- Saving changes
- Loading saved changes
- Playback speed changes
- Notes
- Login and user-locking annotations and files

Items in the tree:

Elements in the tree (in the top left of speechviz) are classes from src/treeClasses.js

There are currently 6 classes.

1. TreeItem which itself isn't meant to be put in the tree, but instead acts as a sort of abstract class for the following classes
2. The second is Popup, which pertains to the popups that occur when you click on an item in the tree to modify something about its contents
3. GroupOfGroups extends TreeItem and represents an element on the tree that contains groups as a child, rather than segments
4. Group extends TreeItem and represents an element on the tree that contains segments as its children
5. Segment extends TreeItem and represents an individual segment of the audio that can be played/looped/removed and shows as a color on peaks
6. Face extends TreeItem and represents an individual Face from face clustering that contains an image of the cluster of faces

If a feature you wish to add to the tree wouldn't work with any of these classes, you should add a class in src/treeClasses.js that extends TreeItem.

Some common things to add that aren't by default gotten from extending TreeItems:

- Icons
    - (play, pause, loop, remove). If you want to reuse the icons from GroupOfGroups/Group (bigger size) add

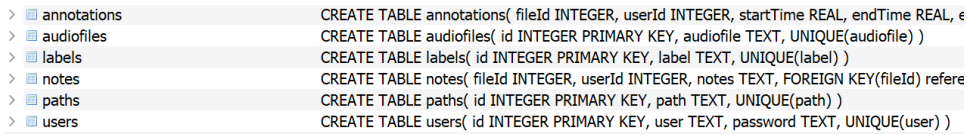
        - static icons = groupIcons;
    - If you want to reuse any of the smaller icons (play, pause, loop, remove, image) from Segment/Face add
        - static icons = segmentIcons;
    - If you want to add icons that are not covered by either of these you can either add the icon you want to add to groupIcons or segmentIcons in the file src/icon.js
        - Find an image that will work in https://feathericons.com/ and add
            - iconName = feather.icons.nameOfIcon.toSvg(options)
            - add name: iconName to segmentIcons or groupIcons
                - or make a new whateverIcons and export and import it, and in src/treeClasses.js in your class do
                    - static icons = whateverIcons;
- Unique constructor
    - Usually takes an id, and customizable options
    - After calling the super constructor, you can also modify what the treeItem constructor did (for example Face gets rid of the play icon made in TreeItem because faces can't be played) or add features unique to the class you're creating

Classes in the tree are actually added to the tree in src/init.js

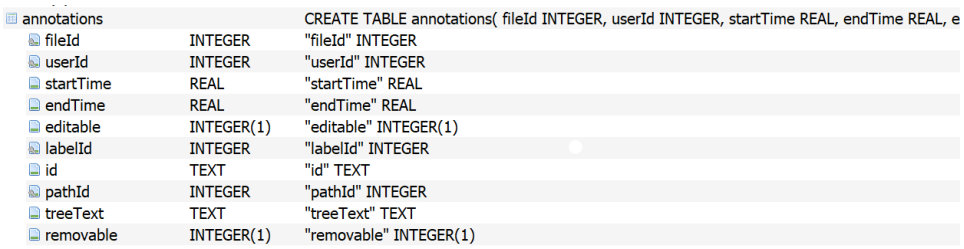
- To add another overarching branch (like Analysis, Clusters) with no parent to the tree
    - const name = new GroupOfGroups("name");
- To add a branch under another
    - const name = new GroupOfGroups("name", { parent: parentName });

Structure of annotation storage:

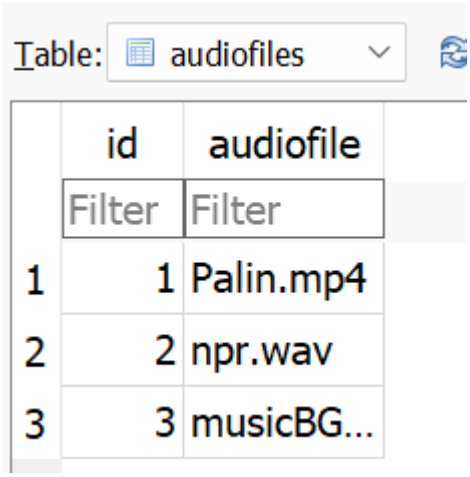
The sqlite database to store annotations is initialized in db_init.py in the main Speechviz folder.



Overall there are 6 tables.

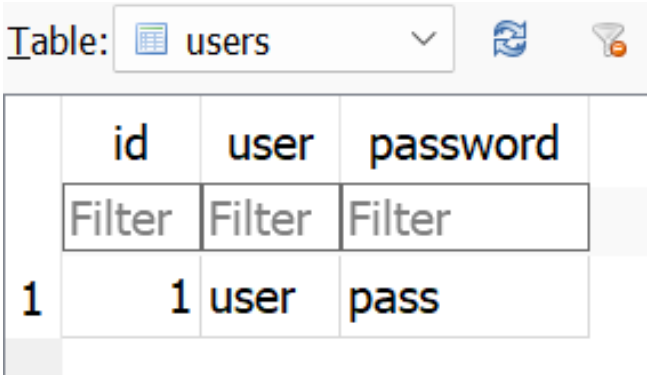


The table that stores information about saved annotations is called annotations. Once initialized it has this structure.



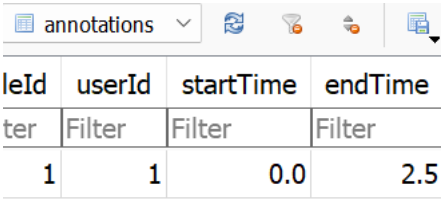
The first piece of information annotations stores is fileId. fileId stores the name of a file as a Primary Key Integer counting up from 1 in the table audiofiles.

This allows you to know which file an annotation belongs to.



The second piece of information userId similarly stores the username and password of a user as a Primary Key Integer counting up from 1 in the table users.

This allows you to know which user did which annotation, and to the annotations you saved while logged in so you can pick up where you left off.



The third piece of information startTime and the fourth piece of information endTime store when a given segment starts and stops respectively.

This allows for to know when the event you're annotating happens, and makes for easy loading.

The fourth piece of information editable, stores whether or not an annotation can be renamed (1 is can be renamed, 0 it cannot). Every segment that is manually added, or automatically generated and then manually moved to a manually created label can be renamed and thus has an editable value of 1. Segments which were automatically generated and moved to another automatically generated label (e.g. a segment from Speaker 1 to Speaker 2).

This helps maintain functionality with the code after being loaded. We can then set the storage in a segment for whether it can or can't be renamed to it's previous value.

Table: labels

	id	label
	Filter	Filter
1	1	Custom Segment 1
2	2	Speaker 1
3	3	jeff

The fifth piece of information annotations stores is labelId. labelId stores the name of a label that a segment that needed to be saved belongs to as a Primary Key Integer counting up from 1 in the table labels.

This allows us to correctly remember the name of a label that was created for a segment, and on load automatically create said label with this segment stored there. Similarly this is the case for automatically generated labels like Speaker 1 when a segment is moved from Speaker to Speaker.

id
Filter
peaks.segment.51
peaks.segment.88
peaks.segment.89
segment.3

The sixth piece of information annotations stores is id. This is what internally we store the segments as. If this id matches the automatically generated segment id, we change its speaker to the label it is stored as having upon loading.

This allows us to automatically change the speaker upon loading with changeSpeaker if it is applicable, as well as not double a segment having it moved to the new Speaker yet remaining at the old speaker.

Table: paths

	id	path
	Filter	Filter
1	1	Segments|Custom-Segments
2	2	Segments|Speakers|Speaker 1
3	3	Segments|Labeled-Speakers|jeff
4	4	Segments|Labeled-Speakers|...
5	5	Segments|Labeled-Speakers|...

The seventh piece of information pathId is stored in the table paths as a primary key Integer counting up from 1. This stores the path a segment has in checkbox tree on the website (separated by |).

This allows us to put the segment in the correct location on the tree and its accuracy is crucial for behaviors with checking, unchecking, moving etc. to function correctly.

removable
Filter
1
1
1
0

The eighth and final piece of information removable is similar to editable, but refers to the ability to delete the segment from the tree with the X button to the right of the play and loop button.

This helps maintain functionality with the code after being loaded.

	fileId	userId	notes
	Filter	Filter	Filter
1	1	1	This is for any notes you wish ...
2	3	1	This is for any notes you wish ...

As an aside the notes taken on speechviz are stored in the last table notes based on the user taking the notes and what file the notes are taken on.

Additionally saving itself, occurs in multiple places in the file, here are the spots.

```
document.querySelector('button[data-action="save"]').addEventListener('click', function () {
  const groupRegex = /Speaker |VAD|Non-VAD/;
  const groups = Object.keys(visibleSegments).filter(group => !group.match(groupRegex));
  let segments = [];
  for (const group of groups) {
    segments = segments.concat(segmentsFromGroup(group, { "visible": true, "hidden": true, "simple": true }));
  }

  let idCounter = highestId + 1;
  segments.map((segment, index) => { return { "index": index, "id": parseInt(segment.id.split(".").at(-1)) }; })
    .sort((seg1, seg2) => seg1.id - seg2.id)
    .map(seg => segments[seg.index])
    .forEach(function (segment) { segment.id = `peaks.segment.${idCounter++}`; });

  const customRegex = /Custom Segment /;
  let numCustom = 1;
  const customChanged = {};
  segments.forEach(function (segment) {
      if (segment.labelText in customChanged) {
        segment.labelText = customChanged[segment.labelText];
      }
      else if (segment.labelText.match(customRegex)) {
        const nextCustom = `Custom Segment ${numCustom++}`;
        customChanged[segment.labelText] = nextCustom;
        segment.labelText = nextCustom;
      }
      if (segment.treeText in customChanged) {
        segment.treeText = customChanged[segment.treeText];
      }
      else if (segment.treeText.match(customRegex)) {
        const nextCustom = `Custom Segment ${numCustom++}`;
        customChanged[segment.treeText] = nextCustom;
        segment.treeText = nextCustom;
      }
    })

  for (const segment of Object.values(moved)) {
    const copied = copySegment(segment, ["color"]);
    copied.path = copied.path.slice(0, -1);
    segments.push(copied);
  }

  const record = { 'user': user, 'filename': fileName, 'segments': segments, "notes": notes.value }
  const json = JSON.stringify(record);
  var request = new XMLHttpRequest();
  request.open('POST', 'save', true);
  request.setRequestHeader('Content-Type', 'application/json; charset=UTF-8');

  request.send(json);
  request.onload = function () {
    // done
    console.log('Annotations saved');
  };
  newChanges = false;
});
```

In speechviz\app.js

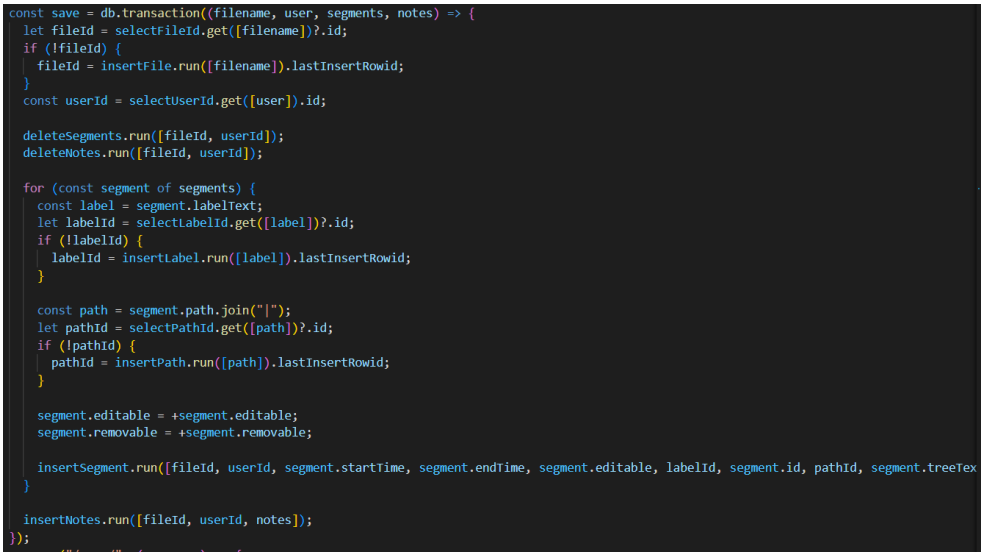
speechviz\src\viz.js

This controls what happens when the save button is pressed. It converts information about segments that need to be saved into a json string to send to be picked up in this next spot.

```
app.use("/save/", (req, res) => {
  save(req.body["filename"], req.body["user"], req.body["segments"], req.body["notes"]);
  res.end();
});
```

In app.js, this info is passed to another function in ap.js, save.

In speechviz\app.js



```
const save = db.transaction((filename, user, segments, notes) => {
  let fileId = selectFileId.get([filename])?.id;
  if (!fileId) {
    fileId = insertFile.run([filename]).lastInsertRowid;
  }
  const userId = selectUserId.get([user]).id;

  deleteSegments.run([fileId, userId]);
  deleteNotes.run([fileId, userId]);

  for (const segment of segments) {
    const label = segment.labelText;
    let labelId = selectLabelId.get([label])?.id;
    if (!labelId) {
      labelId = insertLabel.run([label]).lastInsertRowid;
    }

    const path = segment.path.join("|");
    let pathId = selectPathId.get([path])?.id;
    if (!pathId) {
      pathId = insertPath.run([path]).lastInsertRowid;
    }

    segment.editable = +segment.editable;
    segment.removable = +segment.removable;

    insertSegment.run([fileId, userId, segment.startTime, segment.endTime, segment.editable, labelId, segment.id, pathId, segment.treeTex
  }

  insertNotes.run([fileId, userId, notes]);
});
```

This is what actually puts the information into the database.

How to add annotations with different parameters:

- Go to speechviz\db_init.py

```
# create annotations table
c.execute('''CREATE TABLE IF NOT EXISTS annotations(
    fileId INTEGER,
    userId INTEGER,
    startTime REAL,
    endTime REAL,
    editable INTEGER(1),
    labelId INTEGER,
    id TEXT,
    pathId INTEGER,
    treeText TEXT,
    removable INTEGER(1),
    FOREIGN KEY(fileId) references audiofiles(id),
    FOREIGN KEY(userId) references users(id),
    FOREIGN KEY(labelId) references labels(id),
    FOREIGN KEY(pathId) references paths(id)
)''')
conn.commit()
```

- Add new parameters here, or remove parameters you no longer care about
- Go to where the save button press is triggered speechviz\src\viz.js

```
document.querySelector('button[data-action="save"]').addEventListener('click', function () {
  const groupRegex = /Speaker |VAD|Non-VAD/;
  const groups = Object.keys(visibleSegments).filter(group => !group.match(groupRegex));
  let segments = [];
  for (const group of groups) {
    segments = segments.concat(segmentsFromGroup(group, { "visible": true, "hidden": true, "simple": true }));
  }

  let idCounter = highestId + 1;
  segments.map((segment, index) => { return { "index": index, "id": parseInt(segment.id.split(".").at(-1)) }; })
    .sort((seg1, seg2) => seg1.id - seg2.id)
    .map(seg => segments[seg.index])
    .forEach(function (segment) { segment.id = `peaks.segment.${idCounter++}`; });

  const customRegex = /Custom Segment /;
  let numCustom = 1;
  const customChanged = {};
  segments.forEach(function (segment) {
      if (segment.labelText in customChanged) {
        segment.labelText = customChanged[segment.labelText];
      }
      else if (segment.labelText.match(customRegex)) {
        const nextCustom = `Custom Segment ${numCustom++}`;
        customChanged[segment.labelText] = nextCustom;
        segment.labelText = nextCustom;
      }
      if (segment.treeText in customChanged) {
        segment.treeText = customChanged[segment.treeText];
      }
      else if (segment.treeText.match(customRegex)) {
        const nextCustom = `Custom Segment ${numCustom++}`;
        customChanged[segment.treeText] = nextCustom;
        segment.treeText = nextCustom;
      }
    })

  for (const segment of Object.values(moved)) {
    const copied = copySegment(segment, ["color"]);
    copied.path = copied.path.slice(0, -1);
    segments.push(copied);
  }

  const record = { 'user': user, 'filename': fileName, 'segments': segments, "notes": notes.value }
  const json = JSON.stringify(record);
  var request = new XMLHttpRequest();
  request.open('POST', 'save', true);
  request.setRequestHeader('Content-Type', 'application/json; charset=UTF-8');

  request.send(json);
  request.onload = function () {
    // done
    console.log('Annotations saved');
  };
  newChanges = false;
});
```

- Add the information you wish to send to record (and thus json).
- Modify the save function to add this new information to the database in speechviz\apps.js

```
const save = db.transaction((filename, user, segments, notes) => {
  let fileId = selectFileId.get([filename])?.id;
  if (!fileId) {
    fileId = insertFile.run([filename]).lastInsertRowid;
  }
  const userId = selectUserId.get([user]).id;

  deleteSegments.run([fileId, userId]);
  deleteNotes.run([fileId, userId]);

  for (const segment of segments) {
    const label = segment.labelText;
    let labelId = selectLabelId.get([label])?.id;
    if (!labelId) {
      labelId = insertLabel.run([label]).lastInsertRowid;
    }

    const path = segment.path.join("|");
    let pathId = selectPathId.get([path])?.id;
    if (!pathId) {
      pathId = insertPath.run([path]).lastInsertRowid;
    }

    segment.editable = +segment.editable;
    segment.removable = +segment.removable;

    insertSegment.run([fileId, userId, segment.startTime, segment.endTime, segment.editable, labelId, segment.id, pathId, segment.treeTex
  }

  insertNotes.run([fileId, userId, notes]);
});
```

- Pass the new information into the modified save function

```
app.use("/save/", (req, res) => {
  save(req.body["filename"], req.body["user"], req.body["segments"], req.body["notes"]);
  res.end();
});
```

- Update the loading functionality to load your new information upon launching in speechviz\src\viz.js

```
//#region load annotations
(function () {
  const record = { 'user': user, 'filename': fileName }
  const json = JSON.stringify(record);
  var loadRequest = new XMLHttpRequest();

  loadRequest.open('POST', 'load', true);
  loadRequest.setRequestHeader('Content-Type', 'application/json; charset=UTF-8');

  loadRequest.send(json)
  loadRequest.onload = function () {
    let jsonData = JSON.parse(loadRequest.response);

    notes.value = jsonData.notes || notes.value;

    const regex = /Custom Segment /;
    peaksInstance.segments.add(jsonData.segments, { "overwrite": true }).forEach(function (segment) {
      if (segment.id in segmentsByID) {
        changeSpeaker(peaksInstance, segment.path.concat(segment.id), segmentsByID[segment.id].path, segment);
      }
      else {
        renderSegment(peaksInstance, segment, segment.path.at(-1), segment.path.slice(0, -1));
        sortTree(segment.path.at(-2));
      }
      if (segment.labelText.match(regex)) { segmentCounter++; }
    });

    toggleSegments(peaksInstance, "Segments", false);

    document.getElementById("Segments-nested").classList.add("active");

    groupsCheckboxes["Segments"].checked = true;
    groupsCheckboxes["Segments"].addEventListener("click", function () { toggleSegments(peaksInstance, "Segments", this.checked); });

    segmentsPlay.style.pointerEvents = "auto";
    segmentsLoop.style.pointerEvents = "auto";
    const segmentsPlayIcon = segmentsPlay.firstElementChild;
    const segmentsLoopIcon = segmentsLoop.firstElementChild;
    segmentsPlayIcon.style.stroke = "black";
    segmentsPlayIcon.style.fill = "black";
    segmentsLoopIcon.style.stroke = "black";
    segmentsPlay.addEventListener("click", function () { playGroup(peaksInstance, "Segments"); });
    segmentsLoop.addEventListener("click", function () { playGroup(peaksInstance, "Segments", true); });
  };
})();
```