

Instructions for Web Audio Evaluation Tool

Nicholas Jillings, Brecht De Man and David Moffat

7 December 2015

These instructions are about use of the Web Audio Evaluation Tool [1] on Windows and Mac OS X platforms.

Contents

1	Installation	3
1.1	Contents	3
1.2	Browser	4
2	Test setup	5
2.1	Sample rate	5
2.1.1	Mac OS X	5
2.1.2	Windows	5
2.2	Local test	5
2.2.1	Mac OS X	6
2.2.2	Windows	7
2.3	Remote test	9
3	Using the test create tool	11
3.1	Nodes to familiarise	11
3.2	Modifying <code>core.js</code>	11
3.3	Building the Interface	12
3.3.1	<code>loadInterface</code>	12
3.3.2	<code>loadTest(audioHolderObject)</code>	13
4	Troubleshooting	14
4.1	Opening the JavaScript Console	14

5	Known issues and limitations	16
6	References	17
A	Listening test instructions example	18

1 Installation

Download the folder (<https://code.soundsoftware.ac.uk/hg/webaudioevaluationtool/archive/tip.zip>) and unzip in a location of your choice.

1.1 Contents

The folder should contain the following elements:

Main folder:

- `analyse.html`: analysis and diagnostics of a set of result XML files
- `ape.css`, `core.css`, `graphics.css`, `mushra.css`, `structure.css`: style files (edit to change appearance)
- `ape.js`: JavaScript file for APE-style interface [2]
- `mushra.js`: JavaScript file for MUSHRA-style interface [3]
- `CITING.txt`, `LICENSE.txt`, `README.txt`: text files with, respectively, the citation which we ask to include in any work where this tool or any portion thereof is used, modified or otherwise; the license under which the software is shared; and a general readme file.
- `core.js`: JavaScript file with core functionality
- `index.html`: webpage where interface should appear (includes link to test configuration XML)
- `jquery-2.1.4.js`: jQuery JavaScript Library
- `pythonServer.py`: webserver for running tests locally
- `pythonServer-legacy.py`: webserver with limited functionality (no automatic storing of output XML files)
- `save.php`: PHP script to store result XML files to web server

Documentation (./docs/)

- Instructions: PDF and \LaTeX source of these instructions
- Project Specification Document (\LaTeX /PDF)
- Results Specification Document (\LaTeX /PDF)

- SMC15: PDF and L^AT_EXsource of corresponding SMC2015 publication [1]
- WAC2016: PDF and L^AT_EXsource of corresponding WAC2016 publication

Example project (./example_eval/)

- An example of what the set up XML should look like, with example audio files 0.wav-10.wav which are short recordings at 44.1kHz, 16bit of a woman saying the corresponding number (useful for testing randomisation and general familiarisation with the interface).

Output files (./saves/)

- The output XML files of tests will be stored here by default by the `pythonServer.py` script.

Auxiliary scripts (./scripts/)

- Helpful Python scripts for extraction and visualisation of data.

Test creation tool (./test_create/)

- Webpage for easily setting up your own test without having to delve into the XML.

1.2 Browser

As Microsoft Internet Explorer doesn't support the Web Audio API¹, you will need another browser like Google Chrome, Safari or Firefox (all three are tested and confirmed to work).

The tool is platform-independent and works in any browser that supports the Web Audio API. It does not require any specific, proprietary software. However, in case the tool is hosted locally (i.e. you are not hosting it on an actual webserver) you will need Python (2.7), which is a free programming language - see the next paragraph.

¹<http://caniuse.com/#feat=audio-api>

2 Test setup

2.1 Sample rate

Depending on how the experiment is set up, audio is resampled automatically (the Web Audio default) or the sample rate is enforced. In the latter case, you will need to make sure that the sample rate of the system is equal to the sample rate of these audio files. For this reason, all audio files in the experiment will have to have the same sample rate.

Always make sure that all other digital equipment in the playback chain (clock, audio interface, digital-to-analog converter, ...) is set to this same sample rate.

Note that upon changing the sampling rate, the browser will have to be restarted for the change to take effect.

2.1.1 Mac OS X

To change the sample rate in Mac OS X, go to **Applications/Utilities/Audio MIDI Setup** or find this application with Spotlight (see Figure 1). Then select the output of the audio interface you are using and change the ‘Format’ to the appropriate number. Also make sure the bit depth and channel count are as desired. If you are using an external audio interface, you may have to go to the preference pane of that device to change the sample rate.

Also make sure left and right channel gains are equal, as some applications alter this without changing it back, leading to a predominantly louder left or right channel. See Figure 1 for an example where the channel gains are different.

2.1.2 Windows

To change the sample rate in Windows, right-click on the speaker icon in the lower-right corner of your desktop and choose ‘Playback devices’. Right-click the appropriate playback device and click ‘Properties’. Click the ‘Advanced’ tab and verify or change the sample rate under ‘Default Format’. If you are using an external audio interface, you may have to go to the preference pane of that device to change the sample rate.

2.2 Local test

If the test is hosted locally, you will need to run the local webserver provided with this tool.

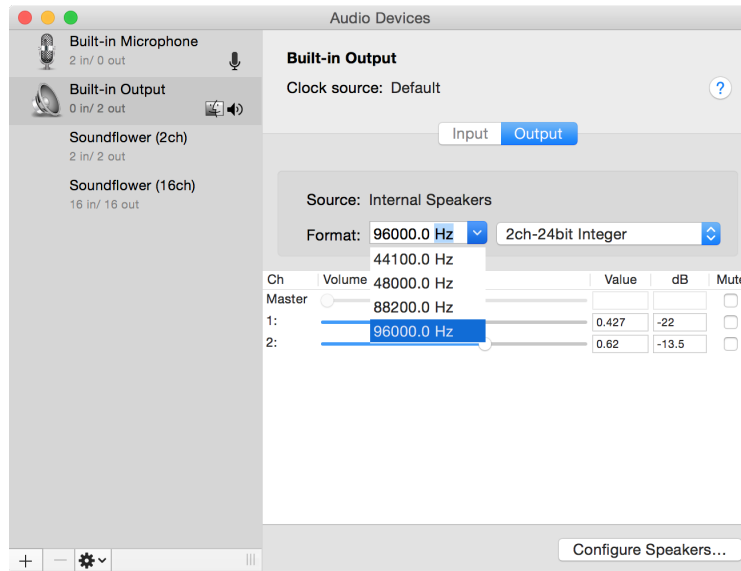


Figure 1: The Audio MIDI Setup window in Mac OS X

2.2.1 Mac OS X

On Mac OS X, Python comes preinstalled.

Open the Terminal (find it in **Applications/Terminal** or via Spotlight), and go to the folder you downloaded. To do this, type `cd [folder]`, where `[folder]` is the folder where to find the `pythonServer.py` script you downloaded. For instance, if the location is `/Users/John/Documents/test/`, then type

```
cd /Users/John/Documents/test/
```

Then hit enter and run the Python script by typing

```
python pythonServer.py
```

and hit enter again. See also Figure 2.

Alternatively, you can simply type `python` (followed by a space) and drag the file into the Terminal window from Finder.

You can leave this running throughout the different experiments (i.e. leave the Terminal open).

To start the test, open the browser and type

```
localhost:8000
```

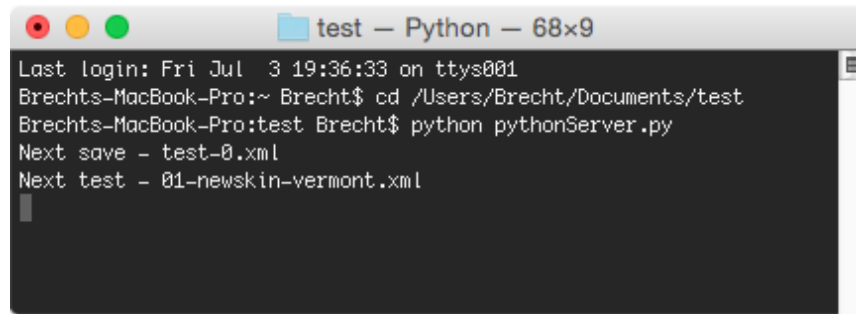


Figure 2: Mac OS X: The Terminal window after going to the right folder (`cd [folder_path]`) and running `pythonServer.py`.

and hit enter. The test should start (see Figure 5).

To quit the server, either close the terminal window or press `Ctrl+C` on your keyboard to forcibly shut the server.

2.2.2 Windows

On Windows, Python 2.7 is not generally preinstalled and therefore has to be downloaded² and installed to be able to run scripts such as the local webserver, necessary if the tool is hosted locally.

Simply double click the Python script `pythonServer.py` in the folder you downloaded.

You may see a warning like the one in Figure 3. Click ‘Allow access’.

The process should now start, in the Command prompt that opens - see Figure 4.

You can leave this running throughout the different experiments (i.e. leave the Command Prompt open).

To start the test, open the browser and type

`localhost:8000`

and hit enter. The test should start (see Figure 5).

If at any point in the test the participant reports weird behaviour or an error of some kind, or the test needs to be interrupted, please notify the experimenter and/or refer to Section 4.

²<https://www.python.org/downloads/windows/>



Figure 3: Windows: Potential warning message when executing `pythonServer.py`.

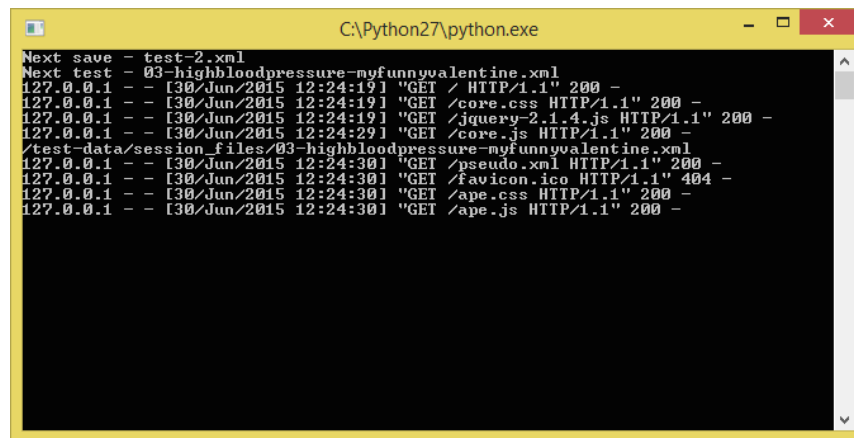


Figure 4: Windows: The Command Prompt after running `pythonServer.py` and opening the corresponding website.

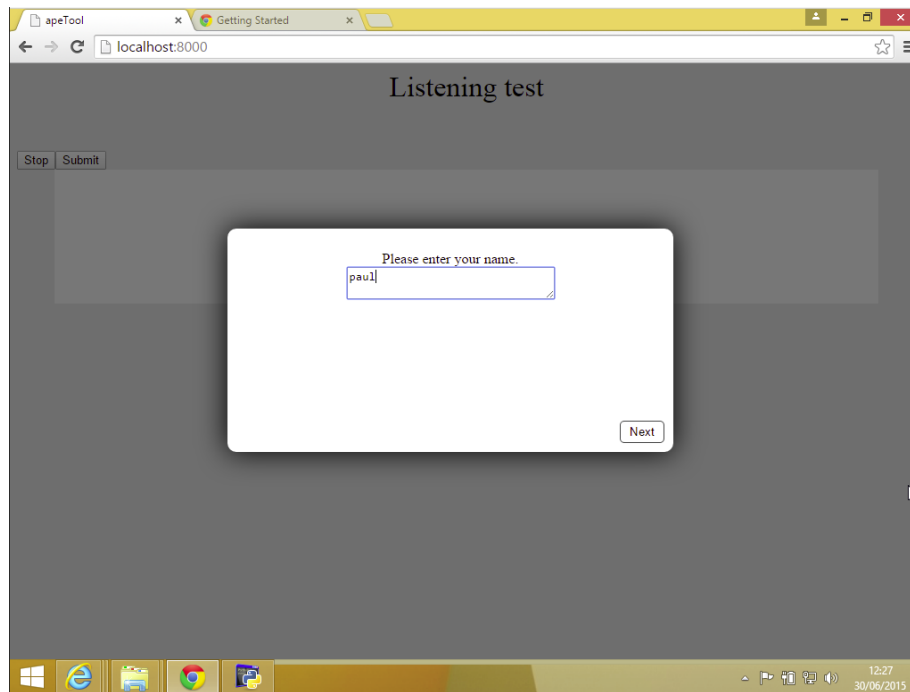


Figure 5: The start of the test in Google Chrome on Windows 7.

When the test is over (the subject should see a message to that effect, and click ‘Submit’ one last time), the output XML file containing all collected data should have appeared in ‘saves/'. The names of these files are ‘test-0.xml’, ‘test-1.xml’, etc., in ascending order. The Terminal or Command prompt running the local web server will display the following file name. If such a file did not appear, please again refer to Section 4.

It is advised that you back up these results as often as possible, as a loss of this data means that the time and effort spent by the subject(s) has been in vain. Save the results to an external or network drive, and/or send them to the experimenter regularly.

To start the test again for a new participant, you do not need to close the browser or shut down the Terminal or Command Prompt. Simply refresh the page or go to `localhost:8000` again.

2.3 Remote test

Put all files on a web server which supports PHP. This allows the ‘save.php’ script to store the XML result files in the ‘saves/’ folder. If the web server is not able to store the XML

file there at the end of the test, it will present the XML file locally to the user, as a ‘Save file’ link.

3 Using the test create tool

We provide a test creation tool, available in the directory `test.create`. This tool is a self-contained web page, so doubling clicking will launch the page in your system default browser.

The test creation tool can help you build a simple test very quickly. By simply selecting your interface and clicking check-boxes you can build a test in minutes.

Include audio by dragging and dropping the stimuli you wish to include.

The tool examines your XML before exporting to ensure you do not export an invalid XML structure which would crash the test.

This guide will help you to construct your own interface on top of the WAET (Web Audio Evaluation Tool) engine. The WAET engine resides in the `core.js` file, this contains prototype objects to handle most of the test creation, operation and data collection. The interface simply has to link into this at the correct points.

3.1 Nodes to familiarise

`Core.js` handles several very important nodes which you should become familiar with. The first is the Audio Engine, initialised and stored in variable `'AudioEngineContext'`. This handles the playback of the web audio nodes as well as storing the `'AudioObjects'`. The `'AudioObjects'` are custom nodes which hold the audio fragments for playback. These nodes also have a link to two interface objects, the comment box if enabled and the interface providing the ranking. On creation of an `'AudioObject'` the interface link will be nulled, it is up to the interface to link these correctly.

The specification document will be decoded and parsed into an object called `'specification'`. This will hold all of the specifications various nodes. The test pages and any pre/post test objects are processed by a test state which will proceed through the test when called to by the interface. Any checks (such as playback or movement checks) are to be completed by the interface before instructing the test state to proceed. The test state will call the interface on each page load with the page specification node.

3.2 Modifying `core.js`

Whilst there is very little code actually needed, you do need to instruct `core.js` to load your interface file when called for from a specification node. There is a function called `'loadProjectSpecCallback'` which handles the decoding of the specification and setting any external items (such as metric collection). At the very end of this function there is an

if statement, add to this list with your interface string to link to the source. There is an example in there for both the APE and MUSHRA tests already included. Note: Any updates to core.js in future work will most likely overwrite your changes to this file, so remember to check your interface is still here after any update that interferes with core.js. Any further files can be loaded here as well, such as css styling files. jQuery is already included.

3.3 Building the Interface

Your interface file will get loaded automatically when the ‘interface’ attribute of the setup node matches the string in the ‘loadProjectSpecCallback’ function. The following functions must be defined in your interface file.

- **loadInterface** - Called once when the document is parsed. This creates any necessary bindings, such as to the metric collection classes and any check commands. Here you can also start the structure for your test such as placing in any common nodes (such as the title and empty divs to drop content into later).
- **loadTest(audioHolderObject)** - Called for each page load. The audioHolderObject contains a specification node holding effectively one of the audioHolder nodes.
- **resizeWindow(event)** - Handle for any window resizing. Simply scale your interface accordingly. This function must be here, but can be an empty function call.

3.3.1 loadInterface

This function is called by the interface once the document has been parsed since some browsers may parse files asynchronously. The best method is simply to put ‘loadInterface()’ at the top of your interface file, therefore when the JavaScript engine is ready the function is called.

By default the HTML file has an element with id “topLevelBody” where you can build your interface. Make sure you blank the contents of that object. This function is the perfect time to build any fixed items, such as the page title, session titles, interface buttons (Start, Stop, Submit) and any holding and structure elements for later on.

At the end of the function, insert these two function calls: `testState.initialise()` and `testState.advanceState();`. This will actually begin the test sequence, including the pre-test options (if any are included in the specification document).

3.3.2 loadTest(audioHolderObject)

This function is called on each new test page. It is this functions job to clear out the previous test and set up the new page. Use the function `audioEngineContext.newTestPage()`; to instruct the audio engine to prepare for a new page. “`audioEngineContext.audioObjects = [];`” will delete any `audioObjects`, `interfaceContext.deleteCommentBoxes()`; will delete any comment boxes and `interfaceContext.deleteCommentQuestions()`; will delete any extra comment boxes specified by `commentQuestion` nodes.

This function will need to instruct the audio engine to build each fragment. Just passing the constructor each element from the `audioHolderObject` will build the track, `audioEngineContext.newTrack(element)` (where `element` is the `audioHolderObject` audio element). This will return a reference to the constructed `audioObject`. Decoding of the audio will happen asynchronously.

You also need to link `audioObject.interfaceDOM` with your interface object for that `audioObject`. The `interfaceDOM` object has a few default methods. Firstly it must start disabled and become enabled once the `audioObject` has decoded the audio (function call: `enable()`). Next it must have a function `exportXMLDOM()`, this will return the xml node for your interface, however the default is for it to return a value node, with `textContent` equal to the normalised value. You can perform other functions, but our scripts may not work if something different is specified (as it will breach our results specifications). Finally it must also have a method `getValue`, which returns the normalised value.

It is also the job the `interfaceDOM` to call any metric collection functions necessary, however some functions may be better placed outside (for example, the APE interface uses drag and drop, therefore the best way was to call the metric functions from the `dragEnd` function, which is called when the interface object is dropped). Metrics based upon listening are handled by the `audioObject`. The `interfaceDOM` object must manage any movement metrics. For a list of valid metrics and their behaviours, look at the project specification document included in the repository/docs location. The same goes for any checks required when pressing the submit button, or any other method to proceed the test state.

4 Troubleshooting

Thanks to feedback from using the interface in experiments by the authors and others, many bugs have been caught and fatal crashes due to the interface (provided it is set up properly by the user) seem to be a thing of the past.

However, if things do go wrong or the test needs to be interrupted for whatever reason, all data is not lost. In a normal scenario, the test needs to be completed until the end (the final ‘Submit’), at which point the output XML is stored in the **saves/**. If this stage is not reached, open the JavaScript Console (see below for how to find it) and type

```
createProjectSave()
```

for a local test or

```
createProjectSave(specification.projectReturn)
```

for a remote test

and hit enter. This will open a pop-up window with a hyperlink that reads ‘Save File’; click it and an XML file with results until that point should be stored in your download folder.

Alternatively, a lot of data can be read from the same console, in which the tool prints a lot of debug information. Specifically:

- the randomisation of pages and fragments are logged;
- any time a slider is played, its ID and the time stamp (in seconds since the start of the test) are displayed;
- any time a slider is dragged and dropped, the location where it is dropped including the time stamp are shown;
- any comments and pre- or post-test questions and their answers are logged as well.

You can select all this and save into a text file, so that none of this data is lost. You may choose to do this even when a test was successful as an extra precaution.

If you encounter any issue which you believe to be caused by any aspect of the , or which the documentation does not mention, please do let us know!

4.1 Opening the JavaScript Console

- In Google Chrome, the JavaScript Console can be found in **View>Developer>JavaScript Console**, or via the keyboard shortcut **Cmd + Alt + J** (Mac OS X).

- In Safari, the JavaScript Console can be found in **Develop>Show Error Console**, or via the keyboard shortcut **Cmd + Alt + C** (Mac OS X). Note that for the Developer menu to be visible, you have to go to Preferences (**Cmd + ,**) and enable ‘Show Develop menu in menu bar’ in the ‘Advanced’ tab. **Note that as long as the Developer menu is not visible, nothing is logged to the console, i.e. you will only be able to see diagnostic information from when you switched on the Developer tools onwards.**
- In Firefox, go to **Tools>Web Developer>Web Console**, or hit **Cmd + Alt + K**.

5 Known issues and limitations

The following is a non-exhaustive list of problems and limitations you may experience using this tool, due to not being supported yet by us, or by the Web Audio API and/or (some) browsers.

- Issue **#1463: Firefox** only supports 8 bit and 16 bit WAV files. Pending automatic requantisation (which deteriorates the audio signal's dynamic range to some extent), WAV format stimuli need to adhere to these limitations in order for the test to be compatible with Firefox.
- Issues **#1474** and **#1462**: On occasions, audio is not working - or only a continuous 'beep' can be heard - notably in **Safari**. Refreshing, quitting the browser and even enabling Developer tools in Safari's Preferences pane ('Advanced' tab: "Show 'Develop' menu in menu bar") has helped resolve this. If no (high quality) audio can be heard, make sure your entire playback system's settings are all correct.

6 References

- [1] N. Jillings, D. Moffat, B. De Man, and J. D. Reiss, “Web Audio Evaluation Tool: A browser-based listening test environment,” in *12th Sound and Music Computing Conference*, July 2015.
- [2] B. De Man and J. D. Reiss, “APE: Audio Perceptual Evaluation toolbox for MATLAB,” in *136th Convention of the Audio Engineering Society*, April 2014.
- [3] *Method for the subjective assessment of intermediate quality level of coding systems*. Recommendation ITU-R BS.1534-1, 2003.

A Listening test instructions example

Before each test, show the instructions below or similar and make sure it is available to the subject throughout the test. Make sure to ask whether the participant has any questions upon seeing and/or reading the instructions.

- You will be asked for your name (“John Smith”) and location (room identifier).
- An interface will appear, where you are asked to
 - click green markers to play the different mixes;
 - drag the markers on a scale to reflect your preference for the mixes;
 - comment on these mixes, using text boxes with corresponding numbers (in your **native language**);
 - optionally comment on all mixes together, or on the song, in ‘General comments’.
- You are asked for your personal, honest opinion. Feel free to use the full range of the scale to convey your opinion of the various mixes. Don’t be afraid to be harsh and direct.
- The markers appear at random positions at first (which means some markers may hide behind others).
- The interface can take a few seconds to start playback, but switching between mixes should be instantaneous.
- This is a research experiment, so please forgive us if things go wrong. Let us know immediately and we will fix it or restart the test.
- When the test is finished (after all songs have been evaluated), just call the experimenter, do NOT close the window.
- After the test, please fill out our survey about your background, experience and feedback on the test.
- By participating, you consent to us using all collected data for research. Unless asked explicitly, all data will be anonymised when shared.

Contact details

- Nicholas Jillings: `nicholas.jillings@mail.bcu.ac.uk`
- Brecht De Man: `b.deman@qmul.ac.uk`
- David Moffat: `d.j.moffat@qmul.ac.uk`