spiegami in modo dettagliato la funzione della seguente porzione di codice:

HTML

``` ``` ``` ```
``` ``` ``` ``` Ready
Play with Effects
```
``` ```
```
Download WAV Download WAV with Effects
Google Drive Authorization ``` Upload WAV to Drive ``` ``` Upload WAV with Effects to Drive ```
Drive: not authorized

CSS
body {
font-family: system-ui, sans-serif;
background: #f6f1e7;
color: #111827;
display: flex;
justify-content: center;
padding: 16px;
margin: 0;
}

.app {
max-width: 780px;
width: 100%;
background: #fbf7f0;
border-radius: 12px;
padding: 16px;
box-shadow: 0 18px 30px rgba(0,0,0,0.25);
border: 1px solid #e5dccb;
}

.section {
margin-bottom: 12px;
border-top: 1px solid #1f2937;
padding-top: 10px;
}
```

```
.waveform-row {
border-top: none;
padding-top: 0;
}

.rec-row {
display: flex;
justify-content: space-between;
gap: 12px;
align-items: center;
}

.block {
display: flex;
flex-wrap: wrap;
align-items: center;
gap: 6px;
}

/* Waveform Canva and Logo */
.waveform-container {
position: relative;
width: 100%;
padding-top: 160px;
}

canvas#waveform {
width: 100%;
height: 150px;
background: #000;
border-radius: 6px;
border: 1px solid #111827;
display: block;
}

.logo {
position: absolute;
top: -60px;
left: 50%;
transform: translateX(-50%);
height: 220px;
pointer-events: none;
z-index: 10;
}

/* Standard Buttons */
button {
background: #2563eb;
```

```
border: none;
color: white;
padding: 6px 10px;
border-radius: 6px;
cursor: pointer;
font-size: 13px;
}
button:disabled {
background: #4b5563;
cursor: not-allowed;
}

/* Rec/Stop Buttons */
.btn-round-rec {
width: 30px;
height: 30px;
border-radius: 50%;
background: #dc2626;
position: relative;
padding: 0;
}
.btn-round-rec::after {
content: "";
position: absolute;
inset: 7px;
background: #fecaca;
border-radius: 50%;
}

.btn-square-stop {
width: 30px;
height: 30px;
border-radius: 8px;
background: #dc2626;
position: relative;
padding: 0;
}
.btn-square-stop::after {
content: "";
position: absolute;
inset: 7px;
background: #fecaca;
border-radius: 4px;
}

#status {
font-size: 11px;
```

```
opacity: 0.8;
}

/* Knobs */
.knobs-grid {
display: grid;
grid-template-columns: repeat(3, minmax(0, 1fr));
gap: 12px;
justify-items: center;
}

.knob-wrapper {
display: flex;
flex-direction: column;
align-items: center;
background: #f3ede2;
border-radius: 10px;
padding: 10px 4px;
border: 1px solid #e0d7c6;
width: 100%;
max-width: 140px;
}

.knob {
width: 60px;
height: 60px;
border-radius: 50%;
background: radial-gradient(circle at 30% 30%, #ffffff, #d9cfbf);
position: relative;
cursor: pointer;
transform: rotate(-135deg);
transition: box-shadow 0.1s ease;
box-shadow: 0 0 0 1px #cfc5b3, 0 6px 14px rgba(0,0,0,0.25);
}
.knob::after {
content: "";
position: absolute;
width: 4px;
height: 22px;
background: #6b5f4a;
border-radius: 4px;
top: 10px;
left: 50%;
transform: translateX(-50%);
}
.knob:active {
```

```
box-shadow: 0 0 0 1px #38bdf8, 0 4px 14px rgba(56,189,248,0.5);
}

.knob-label {
font-size: 12px;
margin-top: 6px;
}

.knob-value {
font-size: 11px;
opacity: 0.8;
}

/* Presets Buttons */
.preset-row {
display: grid;
grid-template-columns: repeat(4, minmax(0, 1fr));
gap: 8px;
border-top: none;
padding-top: 4px;
}

.preset {
background: #10b981;
text-align: center;
width: 100%;
padding: 8px 0;
}

/* Download Buttons */
.download-row {
display: flex;
justify-content: center;
gap: 12px;
}

.download-row button {
min-width: 160px;
}

/* Drive buttons */
.drive-row {
display: flex;
justify-content: center;
gap: 10px;
align-items: center;
flex-wrap: wrap;
}
```

```
.drive-row button {
min-width: 180px;
}

#driveStatus {
justify-content: center;
display: flex;
font-size: 12px;
opacity: 0.8;
}
```

JS

```
//
// ========================================================================
// STATE AND GLOBAL VARIABLES
//
// ========================================================================

// Base recording state
let mediaRecorder, recordedChunks = [], audioBlob = null, audioUrl = null;
let audioCtx, lowpassFilter, highpassFilter, delayNode, delayFeedback, reverbConvolver,
dryGain, wetGain, masterGain, analyser, dataArray, animationId;
let recStartTime = 0;
let recTimerId = null;
let activeProcessedSource = null;
let mediaElementSource = null;

// Google Drive state
const GOOGLE_CLIENT_ID = "704802154881-t0b03q9dc11ijifmopp1f662rnh4hiuf.apps.googleusercontent.com";
const DRIVE_FOLDER_NAME = "Aurora Registrazioni";
const DRIVE_SCOPES = "https://www.googleapis.com/auth/drive.file";

let tokenClient = null;
let driveAccessToken = null;
let driveFolderIdCache = null;

// Waveform canva
const waveformCanvas = document.getElementById("waveform");
const wfCtx = waveformCanvas.getContext("2d");

// Fixed UI elements
const btnStartRec = document.getElementById("btnStartRec");
const btnStopRec = document.getElementById("btnStopRec");
const btnPlayProcessed = document.getElementById("btnPlayProcessed");
const btnDownloadWav = document.getElementById("btnDownloadWav");
const btnDownloadProcessedWav = document.getElementById("btnDownloadProcessedWav");
```

```
const statusEl = document.getElementById("status");
const player = document.getElementById("player");

// Dynamic UI containers
const knobsContainer = document.getElementById("knobsContainer");
const presetsContainer = document.getElementById("presetsContainer");

// Drive UI
const btnAuthDrive = document.getElementById("btnAuthDrive");
const btnUploadWav = document.getElementById("btnUploadWav");
const btnUploadProcessedWav = document.getElementById("btnUploadProcessedWav");
const driveStatusEl = document.getElementById("driveStatus");

// Knobs configuration
const knobsConfig = [
{ id: "gain", label: "Volume", min: 0, max: 100, step: 1, value: 50 },
{ id: "pitch", label: "Pitch", min: 0.5, max: 2.0, step: 0.01, value: 1.00 },
{ id: "lowpass", label: "Lowpass Filter", min: 200, max: 20000, step: 1, value: 20000 },
{ id: "highpass", label: "Highpass Filter",min: 10, max: 5000, step: 1, value: 10 },
{ id: "delayTime", label: "Delay", min: 0, max: 0.5, step: 0.01, value: 0 },
{ id: "reverbMix", label: "Reverb", min: 0, max: 1, step: 0.1, value: 0.3 }
];

// Parameter initial values
const paramValues = {
gain: 0.5,
pitch: 1.00,
lowpass: 20000,
highpass: 10,
delayTime: 0,
reverbMix: 0.3
};

// Presets configuration
const presetsConfig = {
clean: {
label: "Clean",
params: { lowpass: 20000, highpass: 10, delayTime: 0, reverbMix: 0.1, pitch: 1 }
},
phone: {
label: "Phone",
params: { lowpass: 3500, highpass: 400, delayTime: 0, reverbMix: 0.0, pitch: 1 }
},
hall: {
label: "Hall",
params: { lowpass: 18000, highpass: 80, delayTime: 0.25, reverbMix: 0.7, pitch: 1 }
},
lofi: {
```

```
label: "Lo-Fi",
params: { lowpass: 5000, highpass: 150, delayTime: 0.12, reverbMix: 0.4, pitch: 0.9 }
}
};

//
// ========================================================================
// DYNAMIC UI CREATION (KNOBS & PRESETS)
//
// ========================================================================

// Create knob elements dynamically
function createKnobs() {
knobsConfig.forEach(cfg => {
const wrapper = document.createElement("div");
wrapper.className = "knob-wrapper";

  const knob = document.createElement("div");
  knob.className = "knob";
  knob.dataset.target = cfg.id;
  knob.dataset.min = cfg.min;
  knob.dataset.max = cfg.max;
  knob.dataset.step = cfg.step;


  const label = document.createElement("div");
  label.className = "knob-label";
  label.textContent = cfg.label;


  const valueEl = document.createElement("div");
  valueEl.className = "knob-value";
  valueEl.id = cfg.id + "Val";
  valueEl.textContent = (cfg.id === "lowpass" || cfg.id === "highpass")
    ? Math.round(cfg.value) // Condition for frequency knobs (integer values)
    : cfg.value.toFixed(2); // Other knobs (float values with two decimals)


  wrapper.appendChild(knob);
  wrapper.appendChild(label);
  wrapper.appendChild(valueEl);
  knobsContainer.appendChild(wrapper);

});
}

// Create preset buttons dynamically
function createPresets() {
Object.entries(presetsConfig).forEach(([name, preset]) => {
const btn = document.createElement("button");
btn.className = "preset";
```

```
btn.dataset.preset = name;
btn.textContent = preset.label;
btn.addEventListener("click", () ⇒ applyPreset(name));
presetsContainer.appendChild(btn);
});
}

createKnobs();
createPresets();

//
// ================================================================
// AUDIO GRAPH INITIALIZATION
//
// ================================================================

function initAudioGraph() {
if (audioCtx) return;

audioCtx = new (window.AudioContext || window.webkitAudioContext)();

lowpassFilter = audioCtx.createBiquadFilter();
lowpassFilter.type = "lowpass";
lowpassFilter.frequency.value = paramValues.lowpass;

highpassFilter = audioCtx.createBiquadFilter();
highpassFilter.type = "highpass";
highpassFilter.frequency.value = paramValues.highpass;

delayNode = audioCtx.createDelay(5.0);
delayNode.delayTime.value = paramValues.delayTime;

delayFeedback = audioCtx.createGain();
delayFeedback.gain.value = 0.3;
delayNode.connect(delayFeedback);
delayFeedback.connect(delayNode);

reverbConvolver = audioCtx.createConvolver();
reverbConvolver.buffer = createReverbImpulse(audioCtx, 2.5, 2.0);

dryGain = audioCtx.createGain();
wetGain = audioCtx.createGain();
masterGain = audioCtx.createGain();
masterGain.gain.value = paramValues.gain;

const mix = paramValues.reverbMix;
dryGain.gain.value = 1 - mix;
wetGain.gain.value = mix;
```

```
analyser = audioCtx.createAnalyser();
analyser.fftSize = 2048;
dataArray = new Uint8Array(analyser.fftSize);

lowpassFilter.connect(highpassFilter);
highpassFilter.connect(delayNode);
delayNode.connect(dryGain);
delayNode.connect(reverbConvolver);
reverbConvolver.connect(wetGain);
dryGain.connect(masterGain);
wetGain.connect(masterGain);
masterGain.connect(analyser);
analyser.connect(audioCtx.destination);
}

// Connect HTML <audio> element to analyser to draw waveform on raw playback
function connectPlayerToAnalyser() {
if (!audioCtx || !player) return;
if (mediaElementSource) return;
mediaElementSource = audioCtx.createMediaElementSource(player);
mediaElementSource.connect(analyser);
}

// Generate an impulse response for the reverb
function createReverbImpulse(context, duration, decay) {
const rate = context.sampleRate;
const length = rate * duration;
const impulse = context.createBuffer(2, length, rate);
for (let c = 0; c < impulse.numberOfChannels; c++) {
const chData = impulse.getChannelData(c);
for (let i = 0; i < length; i++) {
const n = (length - i) / length;
chData[i] = (Math.random() * 2 - 1) * Math.pow(n, decay);
}
}
return impulse;
}

//
// KNOB BEHAVIOR
//

// Knob utility functions
const lerp = (a, b, t) => a + (b - a) * t; // Linear interpolation for knobs angle-to-value conversion
const clamp = (v, min, max) => Math.min(max, Math.max(min, v)); // To set values within a range
const valueToAngle = (v, min, max) => -135 + ((v - min) / (max - min)) * 270;
```

```
// Knob behavior
const knobElems = document.querySelectorAll(".knob");

knobElems.forEach(knob => {
const id = knob.dataset.target;
const min = +knob.dataset.min; // The + operator is necessary to convert strings to numbers
const max = +knob.dataset.max;
const step = +knob.dataset.step || 0.01; // 0.01 is used as default step value

let value = paramValues[id];
let angle = valueToAngle(value, min, max);
let dragging = false;

knob.style.transform = `rotate(${angle}deg)`; // To let the knobs be in the right default position once the page is loaded

let startMouseAngle = 0;

// Function to get the mouse angle relative to the center of the knob
function mouseAngleDeg(ev, element) {
const r = element.getBoundingClientRect(); // To get dimensions and relative position of the element with respect to the viewport
const cx = r.left + r.width / 2;
const cy = r.top + r.height / 2;
const dx = ev.clientX - cx; // "clientX" and "clientY" give the mouse position relative to the viewport, so this two operations are useful to get the mouse position relative to the center of the knob
const dy = ev.clientY - cy;
return Math.atan2(dy, dx) * (180 / Math.PI);
}

knob.addEventListener("mousedown", (e) => {
e.preventDefault(); // To prevent text selection while dragging ("preventDefault" unables the default browser behavior for the event)
dragging = true;

lastMouseAngle = mouseAngleDeg(e, knob);

document.body.style.userSelect = "none";
});

window.addEventListener("mouseup", () => {
dragging = false;
document.body.style.userSelect = "";
});

window.addEventListener("mousemove", (e) => {
if (!dragging) return;
```

```
const currentMouseAngle = mouseAngleDeg(e, knob);
let delta = currentMouseAngle - lastMouseAngle;

if (delta > 180) delta -= 360;
if (delta < -180) delta += 360;

const speed = e.shiftKey ? 0.35 : 1.0;
angle = clamp(angle + delta * speed, -135, 135);

lastMouseAngle = currentMouseAngle;

knob.style.transform = rotate(${angle}deg);

const t = (angle + 135) / 270;
const raw = lerp(min, max, t);
const v = Math.round(raw / step) * step;

value = v;
updateParam(id, v);
updateValLabel(id, v);
});

updateValLabel(id, value);
updateParam(id, value);
});

// Application of the presets
function applyPreset(name) {
initAudioGraph();

const preset = presetsConfig[name];
if (!preset) return;

const p = preset.params;

const knobs = document.querySelectorAll(".knob");

knobs.forEach(k => {
const id = k.dataset.target;

if (id === "gain") return; // The gain doesn't vary due to the application of the presets

if (!(id in p)) return;

const val = p[id];

paramValues[id] = val; // Update internal state
updateParam(id, val);
updateValLabel(id, val);

const min = +k.dataset.min;
const max = +k.dataset.max;
```

```
k.style.transform = `rotate(${valueToAngle(val, min, max)}deg)`;
});
}

function updateValLabel(id, v) {
const el = document.getElementById(id + "Val");
if (!el) return;
if (id === "lowpass" || id === "highpass") el.textContent = Math.round(v);
else el.textContent = v.toFixed(2);
}

function updateParam(id, v) {
paramValues[id] = v;

if (id === "gain" && masterGain) masterGain.gain.value = v / 100;
else if (id === "lowpass" && lowpassFilter) lowpassFilter.frequency.value = v;
else if (id === "highpass" && highpassFilter) highpassFilter.frequency.value = v;
else if (id === "delayTime" && delayNode) delayNode.delayTime.value = v;
else if (id === "reverbMix" && dryGain && wetGain) {
wetGain.gain.value = v;
dryGain.gain.value = 1 - v;
}
}

//
// ========================================================================
// WAVEFORM
//
// ========================================================================

function drawWaveform() {
if (!analyser) return;
animationId = requestAnimationFrame(drawWaveform);

const w = waveformCanvas.width;
const h = waveformCanvas.height;

analyser.getByteTimeDomainData(dataArray);

wfCtx.fillStyle = "#000";
wfCtx.fillRect(0, 0, w, h);

wfCtx.lineWidth = 2;
wfCtx.strokeStyle = "#38bdf8";
wfCtx.beginPath();

const slice = w / dataArray.length;
let x = 0;
```

for (let i = 0; i < dataArray.length; i++) {
const v = dataArray[i] / 128.0; // Normalize between 0 and 2 to adapt to canvas height (dataArray elements can have values between 0 and 255)
const y = v * h / 2;
if (i === 0) wfCtx.moveTo(x, y);
else wfCtx.lineTo(x, y);
x += slice;
}

wfCtx.lineTo(w, h / 2);
wfCtx.stroke();
}

// Stop waveform animation before leaving the page
window.addEventListener("beforeunload", () ⇒ {
if (animationId) cancelAnimationFrame(animationId);
});

//
// MIC RECORDING
//

btnStartRec.addEventListener("click", async () ⇒ { // async in order to be able to use await inside it
try {
const stream = await navigator.mediaDevices.getUserMedia({ audio: true }); // Mic authorization
mediaRecorder = new MediaRecorder(stream);
recordedChunks = [];

```
mediaRecorder.ondataavailable = e => {
  if (e.data.size) recordedChunks.push(e.data);
};


mediaRecorder.start();
recStartTime = performance.now();


// To update and show the recording timer
if (recTimerId) clearInterval(recTimerId);
recTimerId = setInterval(() => {
const elapsed = (performance.now() - recStartTime) / 1000;
statusEl.textContent = `Recording... ${elapsed.toFixed(1)}s`;
}, 100);


btnStartRec.disabled = true;
```

```
  btnStopRec.disabled = false;
  statusEl.textContent = "Recording...";
```

} catch (e) {
console.error(e);
statusEl.textContent = "Error: the mic is on";
}
});

btnStopRec.addEventListener("click", () => {
mediaRecorder.onstop = () => {
audioBlob = new Blob(recordedChunks, { type: "audio/webm" });
if (audioUrl) URL.revokeObjectURL(audioUrl);
audioUrl = URL.createObjectURL(audioBlob);
player.src = audioUrl;

```
  btnPlayProcessed.disabled = false;
  btnDownloadWav.disabled = false;
  btnDownloadProcessedWav.disabled = false;
  statusEl.textContent = "The recording is ready";


  refreshDriveButtons();
```

};

if (mediaRecorder && mediaRecorder.state === "recording") {
mediaRecorder.stop();
mediaRecorder.stream.getTracks().forEach(t => t.stop()); // Disconnect the mic
}
if (recTimerId) {
clearInterval(recTimerId);
recTimerId = null;
}

btnStartRec.disabled = false;
btnStopRec.disabled = true;
});

//
// PLAYER EVENTS
//

player.addEventListener("play", () => {
initAudioGraph();

```
connectPlayerToAnalyser();
if (!animationId) drawWaveform();

if (activeProcessedSource) {
try {
activeProcessedSource.stop();
} catch (e) {}
activeProcessedSource = null;
}

btnPlayProcessed.disabled = false;
});

player.addEventListener("ended", () => {
if (animationId) {
cancelAnimationFrame(animationId);
animationId = null;
}
});

//
================================================================================
// DOWNLOAD RAW WAV
//
================================================================================

btnDownloadWav.addEventListener("click", () => {
downloadWav({getBlobFn: getRawWavBlob, prefix: "Aurora_"});
});

async function downloadWav({getBlobFn, prefix}) {
if (!audioBlob) return;

try {
const filename = safeTimestampName("wav", prefix);
const wavBlob = await getBlobFn();
downloadBlob(wavBlob, filename);
statusEl.textContent = `Downloaded - ${filename}`;
} catch (e) {
console.error(e);
statusEl.textContent = "Error while exporting the file";
}
}

// To generate the file name with a timestamp
function safeTimestampName(ext, prefix) {
const d = new Date();
const pad = (n) => String(n).padStart(2, "0");
const stamp =
```

```
${d.getFullYear()}-${pad(d.getMonth()+1)}-${pad(d.getDate())}_${pad(d.getHours())}-${pad(d.getMinutes())}-${pad(d.getSeconds())};
return ${prefix}${stamp}.${ext};
}

function downloadBlob(blob, filename) {
const url = URL.createObjectURL(blob);
const a = document.createElement("a");
a.style.display = "none";
a.href = url;
a.download = filename;
document.body.appendChild(a);
a.click();
URL.revokeObjectURL(url);
document.body.removeChild(a);
}

async function getRawWavBlob() {
if (!audioBlob) throw new Error("No recording available");
initAudioGraph();
const arrayBuffer = await audioBlob.arrayBuffer();
const audioBuffer = await audioCtx.decodeAudioData(arrayBuffer);
const wavBuffer = audioBufferToWav(audioBuffer);
return new Blob([wavBuffer], { type: "audio/wav" });
}

//
======================================================================
// PLAYBACK WITH EFFECTS
//
======================================================================

btnPlayProcessed.addEventListener("click", async () => {
if (!audioBlob) return;
initAudioGraph();

// Stop HTML player if it is playing
if (!player.paused) {
player.pause();
player.currentTime = 0;
}

if (activeProcessedSource) {
try {
activeProcessedSource.stop();
} catch (e) {}
activeProcessedSource = null;
}
```

```
const arrayBuffer = await audioBlob.arrayBuffer();
const audioBuffer = await audioCtx.decodeAudioData(arrayBuffer);
const source = audioCtx.createBufferSource();
source.buffer = audioBuffer;
source.playbackRate.value = paramValues.pitch;
source.connect(lowpassFilter);

activeProcessedSource = source;
btnPlayProcessed.disabled = true;

source.onended = () => {
if (activeProcessedSource === source) {
activeProcessedSource = null;
}
btnPlayProcessed.disabled = false;
};

source.start();
if (!animationId) drawWaveform();
});

//
======================================================================
// DOWNLOAD WAV WITH EFFECTS
//
======================================================================

btnDownloadProcessedWav.addEventListener("click", () => {
downloadWav({getBlobFn: getProcessedWavBlob, prefix: "Aurora_fx_"});
});

async function getProcessedWavBlob() {
if (!audioBlob) throw new Error("No recording available");

const arr = await audioBlob.arrayBuffer();
const probeCtx = new (window.AudioContext || window.webkitAudioContext)();
const decoded = await probeCtx.decodeAudioData(arr);
const duration = decoded.duration;
const sampleRate = decoded.sampleRate;
probeCtx.close();

const length = Math.ceil(duration * sampleRate);
const offlineCtx = new (window.OfflineAudioContext || window.webkitOfflineAudioContext)(1,
length, sampleRate);

const source = offlineCtx.createBufferSource();
source.buffer = decoded;
```

```
const lp = offlineCtx.createBiquadFilter();
lp.type = "lowpass";
lp.frequency.value = paramValues.lowpass;

const hp = offlineCtx.createBiquadFilter();
hp.type = "highpass";
hp.frequency.value = paramValues.highpass;

const del = offlineCtx.createDelay(5.0);
del.delayTime.value = paramValues.delayTime;

const fb = offlineCtx.createGain();
fb.gain.value = 0.3;
del.connect(fb);
fb.connect(del);

const conv = offlineCtx.createConvolver();
conv.buffer = createReverbImpulse(offlineCtx, 2.5, 2.0);

const dry = offlineCtx.createGain();
const wet = offlineCtx.createGain();
const master = offlineCtx.createGain();

master.gain.value = paramValues.gain;
const mix = paramValues.reverbMix;
dry.gain.value = 1 - mix;
wet.gain.value = mix;

source.playbackRate.value = paramValues.pitch;

source.connect(lp);
lp.connect(hp);
hp.connect(del);
del.connect(dry);
del.connect(conv);
conv.connect(wet);
dry.connect(master);
wet.connect(master);
master.connect(offlineCtx.destination);

source.start(0);
const rendered = await offlineCtx.startRendering();
const wavBuffer = audioBufferToWav(rendered);
return new Blob([wavBuffer], { type: "audio/wav" });
}

//
==========================================================================
// GOOGLE DRIVE AUTHORIZATION
```

```
//
// ================================================================================

function initDriveAuth() {
if (!btnAuthDrive || !window.google || !google.accounts || !google.accounts.oauth2) { // To verify the existence of the element imported by Google src in the HTML
return;
}
if (tokenClient) return;

// To notify just in case the client ID must be changed or set
if (!GOOGLE_CLIENT_ID || GOOGLE_CLIENT_ID.includes("PASTE_YOUR_CLIENT_ID_HERE")) {
setDriveStatus("Drive: insert GOOGLE_CLIENT_ID in script.js");
btnAuthDrive.disabled = true;
return;
}

btnAuthDrive.addEventListener("click", () => {
setDriveStatus("Drive: authorization in progress...");
tokenClient.requestAccessToken({ prompt: "consent" }); // This function generates the Google consent pop-up
});

tokenClient = google.accounts.oauth2.initTokenClient({client_id: GOOGLE_CLIENT_ID, scope: DRIVE_SCOPES,
callback: (resp) => { // This function is called when requestAccessToken completes and passes the value of resp
driveAccessToken = resp.access_token;
driveFolderIdCache = null; // To force the token to be requested again next time (every time you initialize the web app you have to get Google Drive authorization)
setDriveStatus("Drive: authorized");
refreshDriveButtons();
},
});

}

// To initialize Google Drive authorization immediately after the page is fully loaded
window.addEventListener("load", () => {
initDriveAuth();
// To let the initialization retry in case of failure (due to the async loading)
let tries = 0;
const t = setInterval(() => {
initDriveAuth();
tries++;
if (tokenClient || tries > 40) clearInterval(t);
}, 250);
});
```

```
function setDriveStatus(msg) {
if (driveStatusEl) driveStatusEl.textContent = msg;
}

function refreshDriveButtons() {
const ok = canUploadNow();
if (btnUploadWav) btnUploadWav.disabled = !ok;
if (btnUploadProcessedWav) btnUploadProcessedWav.disabled = !ok;
}

function canUploadNow() {
return !!driveAccessToken && !!audioBlob; // Double negation to convert the element in the
equivalent boolean value
}

//
============================================================================
// GOOGLE DRIVE UPLOAD
//
============================================================================

if (btnUploadWav) {
btnUploadWav.addEventListener("click", async () => {
try {
setDriveStatus("Drive: upload in progress...");
const folderId = await getOrCreateAuroraFolderId();
const wavBlob = await getRawWavBlob();
const filename = safeTimestampName("wav", "Aurora_");
setDriveStatus("Drive: loading WAV...");
const fileId = await uploadBlobToDriveResumable(wavBlob, filename, "audio/wav", folderId);
setDriveStatus(Drive: uploaded - ${filename});
console.log("Drive fileId (mic):", fileId);
} catch (e) {
console.error(e);
setDriveStatus("Drive: upload failed");
}
});
}

if (btnUploadProcessedWav) {
btnUploadProcessedWav.addEventListener("click", async () => {
try {
setDriveStatus("Drive: upload in progress...");
const folderId = await getOrCreateAuroraFolderId();
const wavBlob = await getProcessedWavBlob();
const filename = safeTimestampName("wav", "Aurora_fx_");
setDriveStatus("Drive: loading WAV with effects...");
const fileId = await uploadBlobToDriveResumable(wavBlob, filename, "audio/wav", folderId);
```

```
setDriveStatus(`Drive: uploaded - ${filename}`);
console.log("Drive fileId (fx):", fileId);
} catch (e) {
console.error(e);
setDriveStatus("Drive: upload failed");
}
});
}

async function uploadBlobToDriveResumable(blob, filename, mimeType, folderId) {
const metadata = {
name: filename,
parents: [folderId],
};

const start = await driveFetch("https://www.googleapis.com/upload/drive/v3/files?uploadType=resumable", {
method: "POST",
headers: {
"Content-Type": "application/json; charset=UTF-8",
"X-Upload-Content-Type": mimeType,
"X-Upload-Content-Length": String(blob.size),
},
body: JSON.stringify(metadata),
});

if (!start.ok) throw new Error(await start.text());
const uploadUrl = start.headers.get("Location");
if (!uploadUrl) throw new Error("Missing upload URL");

const put = await fetch(uploadUrl, {
method: "PUT",
headers: {
"Content-Type": mimeType,
"Content-Length": String(blob.size),
},
body: blob,
});

if (!put.ok) throw new Error(await put.text());
const data = await put.json();
return data.id;
}

// To create the Drive folder or get it if it already exists
async function getOrCreateAuroraFolderId() {
if (driveFolderIdCache) return driveFolderIdCache;
```

```
const q = name='${DRIVE_FOLDER_NAME.replace(/'/g, "\\'")}' and mimeType='application/vnd.google-apps.folder' and trashed=false;
const listUrl = https://www.googleapis.com/drive/v3/files?q=${encodeURIComponent(q)}&fields=files(id,name);

const r = await driveFetch(listUrl);
if (!r.ok) throw new Error(await r.text());
const data = await r.json();
if (data.files && data.files.length) {
driveFolderIdCache = data.files[0].id;
return driveFolderIdCache;
}

const createUrl = "https://www.googleapis.com/drive/v3/files";
const body = {
name: DRIVE_FOLDER_NAME,
mimeType: "application/vnd.google-apps.folder",
};
const c = await driveFetch(createUrl, {
method: "POST",
headers: { "Content-Type": "application/json" },
body: JSON.stringify(body),
});
if (!c.ok) throw new Error(await c.text());
const created = await c.json();
driveFolderIdCache = created.id;
return driveFolderIdCache;
}

async function driveFetch(url, options = {}) {
if (!driveAccessToken) throw new Error("Drive not authorized");
const headers = new Headers(options.headers || {});
headers.set("Authorization", Bearer ${driveAccessToken});
return fetch(url, { ...options, headers });
}
```

Ti spiego il codice “a blocchi logici”, così vedi come ogni parte concorre a fare il registratore con effetti e upload su Drive.

Struttura HTML e layout CSS

L’HTML definisce l’interfaccia dell’app “Aurora Audio Recorder”: logo, canvas per la forma d’onda, pulsanti REC/STOP, player, knob degli effetti, preset, download e pulsanti Google Drive. Nel `<head>` vengono caricati: il CSS, la libreria Google Identity (`gsi/client`) per OAuth, la libreria `audiobuffer-to-wav` per convertire AudioBuffer in WAV e il tuo `script.js` con `defer`, così lo script parte solo dopo il parsing dell’HTML.[1]

Il CSS trasforma tutto in una UI “compatta da desktop”:

- `body` centrato con sfondo chiaro; `.app` è il riquadro principale con bordo arrotondato e ombra.
- `.waveform-container` è un contenitore relativo: il `<canvas id="waveform">` fa da display della forma d’onda (sfondo nero), e il logo è assoluto sopra il canvas, centrato con `transform: translateX(-50%)`.
- `.rec-row`, `.download-row`, `.drive-row` sono righe flessibili con `display: flex` o `display: grid` per distribuire pulsanti e blocchi.
- I pulsanti REC/STOP sono stilizzati con `::after`: un cerchio rosso con un pallino chiaro per REC, un quadrato arrotondato per STOP.
- I “knob” sono div e pseudo-elementi CSS: un cerchio con gradiente, un “indicatore” centrale (la lancetta) e una shadow; saranno ruotati via JS per simulare la manopola.

Stato globale, configurazioni e UI dinamica

La parte iniziale del JS definisce lo **stato globale**:

- Variabili per la registrazione: `mediaRecorder`, `recordedChunks`, `audioBlob`, `audioUrl` conservano il risultato della registrazione con MediaRecorder.
- Variabili per il grafo audio Web Audio API: `audioCtx`, i filtri passa-basso e passa-alto, delay, feedback, convolver (riverbero), i gain (dry/wet/master), `analyser` e buffer dati per il disegno della waveform sul canvas.[2]
- Timer di registrazione (`recStartTime`, `recTimerId`) e sorgente audio “processed” (`activeProcessedSource`) per non sovrapporre riproduzioni.
- Stato Google Drive: client ID, nome cartella, scope `drive.file` (accesso solo ai file creati dall’app), token client, token di accesso e cache dell’ID cartella.[3] [4]

Poi vengono recuperati tutti gli elementi HTML con `getElementById` (pulsanti, player, div che conterranno knob e preset, pulsanti Drive, ecc.) e definiti:

- `knobsConfig`: descrive ogni manopola (id logico, label, min/max/step, valore iniziale).
- `paramValues`: oggetto che mantiene i valori correnti delle manopole (gain, pitch, lowpass, ecc.).
- `presetsConfig`: preset con un’etichetta e un sotto-oggetto `params` che sovrascrive alcune manopole (es. “Phone” taglia basse e alte per effetto telefono).

Creazione dinamica di knob e preset

`createKnobs()` cicla `knobsConfig` e per ogni config crea:

- un `div.knob-wrapper`;
- un `div.knob` (solo grafica, senza `<input>`), con `dataset` per id, min, max, step;
- una label (`.knob-label`) e un display del valore (`.knob-value`, con id tipo `"gainVal"`).

Il valore di testo è formattato: per lowpass/highpass come intero arrotondato, per gli altri con due decimali.

`createPresets()` cicla `presetsConfig`, crea un bottone per preset, imposta `data-preset` e `textContent`, e gli aggancia un click handler che chiama `applyPreset(name)`.

Alla fine: `createKnobs(); createPresets();` genera l'intera UI di controllo a runtime, senza scrivere i knob nel markup.

Grafo audio Web Audio API ed effetti

`initAudioGraph()` inizializza una sola volta l'AudioContext e tutta la catena di effetti:

1. Crea `audioCtx = new (window.AudioContext || window.webkitAudioContext)()`.
2. Crea `lowpassFilter` e `highpassFilter` (BiquadFilter) e imposta le frequenze iniziali da `paramValues`.
3. Crea `delayNode` con max 5 s e un `delayFeedback` che fa il loop (delay → feedback gain → delay) per l'eco.
4. Crea il `reverbConvolver` e gli assegna un impulse response sintetico tramite `createReverbImpulse(audioCtx, 2.5, 2.0)` (rumore decrescente su 2 canali).
5. Crea `dryGain`, `wetGain` e `masterGain`; `masterGain.gain` viene impostato da `paramValues.gain`. Il mix dry/wet è deciso da `reverbMix`: `dry=1-mix`, `wet=mix`.
6. Crea `analyser` per estrarre i campioni time-domain e disegnare la waveform; `fftSize = 2048` prepara un buffer di 2048 campioni.[2]

La catena è:

`lowpass → highpass → delay → (dryGain + reverb → wetGain) → masterGain → analyser → audioCtx.destination`.

`connectPlayerToAnalyser()` collega la `<audio id="player">` al grafo tramite `createMediaElementSource(player)` che va nell`analyser`, così la waveform segue il playback del file grezzo.[2]

`createReverbImpulse(context, duration, decay)` genera un `AudioBuffer` stereo e riempie ogni canale con noise attenuato nel tempo (`Math.pow(n, decay)`), simulando un riverbero a coda esponenziale.

Logica delle manopole (drag angolare → valore parametro)

Per tutte le `.knob`:

- Recupera `id`, `min`, `max`, `step` dai dataset e il valore iniziale da `paramValues`.
- Con `valueToAngle(v, min, max)` converte il valore in angolo tra -135° e +135°; il knob parte ruotato in quella posizione.
- Gestisce il drag con eventi `mousedown`, `mousemove`, `mouseup`:

- Su `mousedown`: `dragging = true`, calcola `lastMouseAngle` rispetto al centro del knob con `mouseAngleDeg`, disabilita la selezione di testo (userSelect "none").
- Su `mousemove`: se `dragging`, calcola `currentMouseAngle`, ricava `delta` (corretto per cross 180°/−180°), moltiplica per `speed` (più preciso se `Shift` premuto), aggiorna `angle = clamp(angle + delta*speed, -135, 135)`.
- Aggiorna `lastMouseAngle`, ruota il knob via `style.transform = rotate(...)`, calcola il valore `v` tramite interpolazione lineare (`lerp(min, max, t)` con `t` normalizzato sull'intervallo di angolo), quantizza su `step`, aggiorna `paramValues` e UI (`updateParam` e `updateValLabel`).

In questo modo la manopola è completamente controllata via coordinate del mouse e trigonometria, senza slider HTML.

`applyPreset(name)`:

- Assicura che il grafo audio sia inizializzato.
- Recupera il preset, e per ogni knob esistente:
  - ignora il gain (volume non cambia con i preset);
  - se c'è un valore nel preset, aggiorna `paramValues`, chiama `updateParam` (per cambiare il nodo Web Audio) e `updateValLabel` (per la label di testo).
  - aggiorna la rotazione del knob usando `valueToAngle`.

`updateParam(id, v)` è il punto di contatto UI → audio engine:

- `gain`: se `masterGain` esiste, imposta `masterGain.gain.value = v / 100`.
- `lowpass`: aggiorna `lowpassFilter.frequency.value`.
- `highpass`: aggiorna `highpassFilter.frequency.value`.
- `delayTime`: aggiorna `delayNode.delayTime.value`.
- `reverbMix`: aggiorna `wetGain.gain = v` e `dryGain.gain = 1 - v`.

Disegno della waveform

`drawWaveform()` è un loop animato che usa `requestAnimationFrame`:

- Se `analyser` non esiste, esce. Altrimenti richiede il prossimo frame e memorizza `animationId`.
- Legge la larghezza/altezza del canvas e riempie lo sfondo nero (`fillRect`).
- Usa `analyser.getByteTimeDomainData(dataArray)` per riempire `dataArray` con campioni tra 0 e 255.[2]
- Calcola la distanza orizzontale `slice = w / dataArray.length`, e poi, per ogni campione, normalizza `v = dataArray[i]/128` e mappa sulla metà altezza (`y = v * h/2`), tracciando una linea continua.
- Completa con una linea fino al centro dell'asse (`h/2`) e `stroke` per disegnare.

Il listener `beforeunload` cancella l'animazione alla chiusura della pagina.

Registrazione microfono (MediaRecorder)

Sul click di `btnStartRec`:

- Chiede accesso al microfono con `navigator.mediaDevices.getUserMedia({ audio: true })`, che ritorna una `MediaStream` in un contesto sicuro (https).
- Crea `mediaRecorder = new MediaRecorder(stream)`, svuota `recordedChunks` e definisce `ondataavailable` per pushare i chunk nel vettore quando disponibili.[5] [6] [7] [2]
- `mediaRecorder.start();` marca il tempo con `performance.now()`.
- Imposta un `setInterval` ogni 100 ms per aggiornare `statusEl` con il tempo trascorso in secondi (timer di registrazione).
- Disabilita il pulsante Start, abilita Stop e aggiorna lo status.

Sul click di `btnStopRec`:

- Definisce `mediaRecorder.onstop` per costruire il `Blob` audio da `recordedChunks` (`type: "audio/webm"`), crea una `ObjectURL` e la mette come `src` del player `<audio>`.
- Abilita: `Play with Effects`, `Download WAV` e `Download WAV with Effects`, aggiorna lo status e aggiorna i pulsanti Drive con `refreshDriveButtons()`.
- Ferma la registrazione se lo stato è `"recording"`, chiude i track dello stream (stop microfono), cancella il timer, riabilita Start e disabilita Stop.

Player eventi e WAV RAW

Eventi sul player:

- `player.addEventListener("play", ...)`: inizializza grafo audio, collega il player all'analyser e fa partire `drawWaveform()` se non già attivo. Se c'è una sorgente "processed" in riproduzione (`activeProcessedSource`), la ferma per evitare audio sovrapposto.
- `player.addEventListener("ended", ...)`: alla fine del file, ferma l'animazione.

Download WAV grezzo:

- `btnDownloadWav` chiama `downloadWav({getBlobFn: getRawWavBlob, prefix: "Aurora_"})`.
- `downloadWav` genera un nome file con timestamp (`safeTimestampName`), chiama `getBlobFn()` per ottenere un Blob WAV e lo scarica tramite `downloadBlob` (crea `<a>`, `href` a `createObjectURL`, `click()`, poi revoca URL).
- `getRawWavBlob()`:
  - controlla che esista `audioBlob`;
  - inizializza l'audio graph;
  - legge il Blob (`arrayBuffer`), lo decodifica in `AudioBuffer` con `audioCtx.decodeAudioData`, lo passa a `audioBufferToWav` (fornita dalla libreria caricata da CDN) e ritorna un Blob `audio/wav`.[1]

Playback con effetti (in tempo reale)

`btnPlayProcessed`:

- Controlla che ci sia `audioBlob`. Inizializza l'audio graph.
- Se il player HTML sta suonando, lo ferma e resetta `currentTime`.
- Se una precedente sorgente processed è ancora attiva, la ferma.
- Decodifica `audioBlob` in `AudioBuffer`, crea `audioCtx.createBufferSource()`, imposta `source.buffer`, configura `source.playbackRate.value = paramValues.pitch` (pitch shifting rudimentale per time-stretching).
- Collega la sorgente direttamente all'inizio della catena effetti (`lowpassFilter`).
- Memorizza `activeProcessedSource = source`, disabilita il pulsante finché la riproduzione non termina, definisce `onended` per riabilitarlo.
- `source.start()` avvia la riproduzione, e se la waveform non è in corso, chiama `drawWaveform()`.

Render offline e download WAV "con effetti"

`btnDownloadProcessedWav` usa `downloadWav` ma con `getProcessedWavBlob` e prefisso `"Aurora_fx_"`.

`getProcessedWavBlob()`:

1. Verifica `audioBlob`.
2. Usa un `AudioContext` temporaneo (`probeCtx`) per decodificare il blob e ricavare `duration` e `sampleRate`, poi chiude il context.
3. Crea un `OfflineAudioContext` mono con lunghezza corretta `length = ceil(duration*sampleRate)`. L'OfflineAudioContext permette di renderizzare l'audio con effetti in modo deterministico, senza dipendere dal tempo reale.[2]
4. Crea un `BufferSource`, filtri lowpass/highpass, delay + feedback, convolver (con impulse `createReverbImpulse(offlineCtx, ...)`), gain dry/wet e master, come nella versione realtime.
5. Imposta `source.playbackRate.value = paramValues.pitch`.
6. Connette la catena e `master` al `offlineCtx.destination`, `source.start(0)` e attende `offlineCtx.startRendering()`.
7. Converte il `AudioBuffer` renderizzato in WAV con `audioBufferToWav` e ritorna un Blob `audio/wav`.[1]

Così il file scaricato contiene esattamente gli effetti impostati dalle manopole.

Integrazione Google Drive (OAuth e upload)

Inizializzazione OAuth

`initDriveAuth()`:

- Controlla che il bottone esista e che la libreria `google.accounts.oauth2` sia disponibile (caricata dallo script nel `<head>`).
- Se `GOOGLE_CLIENT_ID` manca o è placeholder, scrive un messaggio e disabilita il pulsante.
- Aggancia un click listener a `btnAuthDrive`: cambia lo status e chiama `tokenClient.requestAccessToken({ prompt: "consent" })`, che mostra il popup di consenso Google.
- Inizializza `tokenClient = google.accounts.oauth2.initTokenClient({ ... })` con:
  - `client_id: GOOGLE_CLIENT_ID`;
  - `scope: DRIVE_SCOPES` (solo `drive.file`);
  - `callback`: quando l'utente concede l'accesso, ricevi `resp.access_token`, lo salvi in `driveAccessToken`, resetti la cache cartella e abiliti i pulsanti upload.[4] [3]

Un listener `window.addEventListener("load", ...)` chiama `initDriveAuth()` appena la pagina è pronta, e un `setInterval` di retry (fino a 40 volte ogni 250 ms) copre il caso in cui lo script Google arrivi in ritardo.

Stato Drive e pulsanti

- `setDriveStatus(msg)` aggiorna il testo sotto i pulsanti Drive.
- `canUploadNow()` torna `true` solo se hai sia un `driveAccessToken` che un `audioBlob`.
- `refreshDriveButtons()` abilita/disabilita i pulsanti upload in base a `canUploadNow()`.

Upload dei file

Due listener:

- `btnUploadWav`:
  - Aggiorna stato ("upload in progress…");
  - `getOrCreateAuroraFolderId()` ottiene o crea la cartella `Aurora Registrazioni` su Drive;
  - `getRawWavBlob()` produce il WAV grezzo;
  - genera filename con timestamp;
  - aggiorna stato ("loading WAV…");
  - chiama `uploadBlobToDriveResumable()` per fare upload con protocollo resumable;
  - aggiorna status con "uploaded – nomefile" e logga l'id.
- `btnUploadProcessedWav`: uguale ma con `getProcessedWavBlob()` e prefix `"Aurora_fx_"`.

`uploadBlobToDriveResumable(blob, filename, mimeType, folderId)`:

1. Prepara metadati `{name, parents:[folderId]}`.
2. Prima richiesta `POST` a `https://www.googleapis.com/upload/drive/v3/files?uploadType=resumable` con JSON metadati e header "X-Upload-Content-*"; usa `driveFetch` per includere automaticamente il Bearer token.
3. Se ok, legge header `Location` che contiene l'URL della sessione di upload.
4. Seconda richiesta `PUT` all'URL di upload con il blob WAV.
5. Se ok, parse del JSON e ritorna `data.id` (id file su Drive).[4]

`getOrCreateAuroraFolderId()`:

- Se `driveFolderIdCache` è già valorizzato, la riusa.
- Altrimenti interroga Drive con query `name='Aurora Registrazioni' and mimeType='application/vnd.google-apps.folder' and trashed=false` sull'endpoint `files` filtrando `fields=files(id,name)`.
- Se esiste almeno una cartella, memorizza e ritorna l'id della prima.
- Se non c'è, fa un `POST` a `https://www.googleapis.com/drive/v3/files` con body `{name: DRIVE_FOLDER_NAME, mimeType:"application/vnd.google-apps.folder"}` per crearla, memorizza l'id e lo ritorna.[4]

`driveFetch(url, options)` è un wrapper su `fetch` che aggiunge sempre l'header `Authorization: Bearer <driveAccessToken>`, e lancia errore se il token non c'è.

Se vuoi, nel prossimo messaggio posso prendere una porzione specifica (es. solo knob, o solo upload Drive) e riscriverla con commenti ancora più "a basso livello" riga per riga.

⁂

Si, fallo per tutte le sezioni del codice JS

Vado sezione per sezione del JS e ti riscrivo il comportamento "riga per riga" (o piccolo blocco per blocco).

Stato globale e configurazioni iniziali

```
let mediaRecorder, recordedChunks = [], audioBlob = null, audioUrl = null;
```

- Dichiara `mediaRecorder` (sarà l'istanza `MediaRecorder`), un array `recordedChunks` per accumulare i chunk di dati registrati, e due variabili per contenere il Blob finale (`audioBlob`) e il relativo URL (`audioUrl`).[11] [12]

```
let audioCtx, lowpassFilter, highpassFilter, delayNode, delayFeedback, reverbConvolver, d
```

- Dichiara l'`AudioContext` (motore audio) e tutti i nodi della catena effetti (filtri, delay, convolver per il riverbero, guadagni dry/wet/master, `analyser` per disegnare waveform, array dati e id

dell'animazione).[13]

```
let recStartTime = 0;
let recTimerId = null;
```

- Memorizza l'istante d'inizio registrazione (millisecondi) e l'id del timer che aggiorna il display del tempo.

```
let activeProcessedSource = null;
let mediaElementSource = null;
```

- `activeProcessedSource` tiene la sorgente Web Audio che riproduce l'audio "con effetti", così puoi fermarla se necessario.
- `mediaElementSource` terrà il nodo che collega `<audio>` all'`analyser`.

Stato Google Drive

```
const GOOGLE_CLIENT_ID = "...";
const DRIVE_FOLDER_NAME = "Aurora Registrazioni";
const DRIVE_SCOPES = "https://www.googleapis.com/auth/drive.file";
```

- Configurazioni costanti: client ID OAuth, nome cartella su Drive, e scope `drive.file` (accesso solo ai file creati/modificati dall'app).[14]

```
let tokenClient = null;
let driveAccessToken = null;
let driveFolderIdCache = null;
```

- `tokenClient`: oggetto restituito da `google.accounts.oauth2.initTokenClient`.
- `driveAccessToken`: conterrà l'access token OAuth per Drive.
- `driveFolderIdCache`: memorizza l'id della cartella "Aurora Registrazioni" dopo la prima volta, per non rifare query ogni volta.[15] [14]

Riferimenti al DOM

```
const waveformCanvas = document.getElementById("waveform");
const wfCtx = waveformCanvas.getContext("2d");
```

- Trova il `<canvas id="waveform">` e ne prende il contesto 2D per disegnare la forma d'onda.

Poi, una serie di `getElementById`:

```
const btnStartRec = document.getElementById("btnStartRec");
...
const player = document.getElementById("player");
```

```
...
const knobsContainer = document.getElementById("knobsContainer");
...
const presetsContainer = document.getElementById("presetsContainer");
...
const btnAuthDrive = document.getElementById("btnAuthDrive");
...
const driveStatusEl = document.getElementById("driveStatus");
```

- Ogni costante punta a un elemento UI (pulsanti, player audio, contenitori per knob e preset, pulsanti Drive, span di stato).

Configurazione knob, parametri e preset

```
const knobsConfig = [
  { id: "gain",      label: "Volume",  min: 0,   max: 100,   step: 1,    value: 50 },
  ...
];
```

- È una "tabella" che descrive le manopole: nome logico (`id`), etichetta, range min/max, passo di quantizzazione e valore iniziale.

```
const paramValues = {
  gain: 0.5,
  pitch: 1.00,
  lowpass: 20000,
  highpass: 10,
  delayTime: 0,
  reverbMix: 0.3
};
```

- Mappa dei valori correnti dei parametri audio: è lo stato "sorgente di verità" che le manopole e i nodi audio leggono/aggiornano.

```
const presetsConfig = {
  clean: { label: "Clean", params: {...} },
  phone: { ... },
  hall: { ... },
  lofi:  { ... }
};
```

- Ogni preset ha un nome (chiave), una label da mostrare sul bottone e un oggetto `params` con solo i parametri da sovrascrivere (non tocca `gain`).

Creazione dinamica dei knob

```
function createKnobs() {
  knobsConfig.forEach(cfg => {
    const wrapper = document.createElement("div");
    wrapper.className = "knob-wrapper";
```

- Crea un contenitore `div` per il knob (con label e valore).

```
    const knob = document.createElement("div");
    knob.className = "knob";
    knob.dataset.target = cfg.id;
    knob.dataset.min = cfg.min;
    knob.dataset.max = cfg.max;
    knob.dataset.step = cfg.step;
```

- Crea la “manopola” grafica vera e propria, e usando `dataset` memorizza parametri utili per il controllo (id, min, max, step).

```
    const label = document.createElement("div");
    label.className = "knob-label";
    label.textContent = cfg.label;
```

- Div per l’etichetta testuale (es. “Volume”).

```
    const valueEl = document.createElement("div");
    valueEl.className = "knob-value";
    valueEl.id = cfg.id + "Val";
```

- Div che mostra il valore numerico corrente, con id tipo `"gainVal"` per poterlo aggiornare dopo.

```
    valueEl.textContent = (cfg.id === "lowpass" || cfg.id === "highpass")
      ? Math.round(cfg.value)
      : cfg.value.toFixed(2);
```

- Imposta il testo iniziale: se è un filtro (frequenze) mostra intero, altrimenti due decimali.

```
    wrapper.appendChild(knob);
    wrapper.appendChild(label);
    wrapper.appendChild(valueEl);
    knobsContainer.appendChild(wrapper);
  });
}
```

- Assembla tutto e append il wrapper nel contenitore centrale dei knob.

Creazione dinamica dei preset

```
function createPresets() {
  Object.entries(presetsConfig).forEach(([name, preset]) => {
    const btn = document.createElement("button");
    btn.className = "preset";
    btn.dataset.preset = name;
    btn.textContent = preset.label;
```

- Per ogni preset crea un bottone, gli assegna una classe, un attributo `data-preset` col nome logico e un testo leggibile.

```
    btn.addEventListener("click", () => applyPreset(name));
    presetsContainer.appendChild(btn);
  });
}
```

- Al click del bottone richiama `applyPreset` con il nome del preset, e infine aggiunge il bottone al DOM.

```
createKnobs();
createPresets();
```

- Richiama subito entrambe le funzioni per costruire UI knob + preset all'avvio.

Inizializzazione grafo audio

```
function initAudioGraph() {
  if (audioCtx) return;
```

- Se `audioCtx` è già stato creato, non fa nulla (evita re-inizializzazioni).

```
  audioCtx = new (window.AudioContext || window.webkitAudioContext)();
```

- Crea un `AudioContext` compatibile (anche su Safari usando `webkitAudioContext`).[13]

```
  lowpassFilter = audioCtx.createBiquadFilter();
  lowpassFilter.type = "lowpass";
  lowpassFilter.frequency.value = paramValues.lowpass;
```

- Crea il filtro passa-basso, ne imposta il tipo e la frequenza iniziale.

```
  highpassFilter = audioCtx.createBiquadFilter();
  highpassFilter.type = "highpass";
  highpassFilter.frequency.value = paramValues.highpass;
```

- Filtro passa-alto, con frequenza iniziale.

```
delayNode = audioCtx.createDelay(5.0);
delayNode.delayTime.value = paramValues.delayTime;
```

- Crea un `DelayNode` con max delay 5 s, e imposta il valore iniziale (in secondi).

```
delayFeedback = audioCtx.createGain();
delayFeedback.gain.value = 0.3;
delayNode.connect(delayFeedback);
delayFeedback.connect(delayNode);
```

- Crea un nodo Gain di feedback, con guadagno 0.3, e chiude l'anello `delay → feedback → delay` per creare ripetizioni.

```
reverbConvolver = audioCtx.createConvolver();
reverbConvolver.buffer = createReverbImpulse(audioCtx, 2.5, 2.0);
```

- Crea un Convolver (per riverbero) e gli assegna un impulse response sintetico generato dalla funzione sotto.

```
dryGain = audioCtx.createGain();
wetGain = audioCtx.createGain();
masterGain = audioCtx.createGain();
masterGain.gain.value = paramValues.gain;
```

- Crea tre nodi Gain: "dry" (suono diretto), "wet" (riverbero) e master (volume complessivo), settando quest'ultimo dal valore corrente.

```
const mix = paramValues.reverbMix;
dryGain.gain.value = 1 - mix;
wetGain.gain.value = mix;
```

- Calcola il mix: più `mix` alto, più riverbero (wet), meno segnale diretto (dry).

```
analyser = audioCtx.createAnalyser();
analyser.fftSize = 2048;
dataArray = new Uint8Array(analyser.fftSize);
```

- `AnalyserNode` per leggere i campioni in time-domain; imposta `fftSize` e crea un buffer di byte per i dati.[13]

```
lowpassFilter.connect(highpassFilter);
highpassFilter.connect(delayNode);
delayNode.connect(dryGain);
delayNode.connect(reverbConvolver);
reverbConvolver.connect(wetGain);
dryGain.connect(masterGain);
wetGain.connect(masterGain);
masterGain.connect(analyser);
```

```
  analyser.connect(audioCtx.destination);
}
```

- Definisce la catena: filtri → delay → (dry + riverbero) → master → analyser → uscita audio (casse).

Collegare l'elemento `<audio>` alla waveform

```
function connectPlayerToAnalyser() {
  if (!audioCtx || !player) return;
  if (mediaElementSource) return;
  mediaElementSource = audioCtx.createMediaElementSource(player);
  mediaElementSource.connect(analyser);
}
```

- Se il contesto audio esiste e il player esiste, e non è già stato creato un source, crea un `MediaElementSource` dal tag `<audio>` e lo collega all'`analyser` (oltre al suo routing interno).

Creazione impulse per riverbero

```
function createReverbImpulse(context, duration, decay) {
  const rate = context.sampleRate;
  const length = rate * duration;
  const impulse = context.createBuffer(2, length, rate);
```

- Calcola quanti campioni servono per la durata desiderata, crea un `AudioBuffer` stereo (`2` canali) di quella lunghezza e sample rate.

```
  for (let c = 0; c < impulse.numberOfChannels; c++) {
    const chData = impulse.getChannelData(c);
    for (let i = 0; i < length; i++) {
      const n = (length - i) / length;
      chData[i] = (Math.random() * 2 - 1) * Math.pow(n, decay);
    }
  }
  return impulse;
}
```

- Per ogni canale, riempie l'array con valori casuali da -1 a 1, ma modulati da `Math.pow(n, decay)` per ottenere una coda che svanisce nel tempo (più "riverberosa").

Comportamento dei knob

Utility:

```
const lerp = (a, b, t) => a + (b - a) * t;
const clamp = (v, min, max) => Math.min(max, Math.max(min, v));
const valueToAngle = (v, min, max) => -135 + ((v - min) / (max - min)) * 270;
```

- `lerp`: interpolazione lineare tra `a` e `b` in base a `t` (0–1).
- `clamp`: limita `v` dentro [min, max].
- `valueToAngle`: trasforma un valore nel range [min, max] in un angolo tra -135° e +135°.

```
const knobElems = document.querySelectorAll(".knob");
```

- Seleziona tutti gli elementi knob creati dinamicamente.

Per ogni knob:

```
knobElems.forEach(knob => {
  const id = knob.dataset.target;
  const min = +knob.dataset.min;
  const max = +knob.dataset.max;
  const step = +knob.dataset.step || 0.01;
```

- Recupera id logico e range dai dataset; `+` converte da stringa a numero. Se lo step non c'è, default 0.01.

```
  let value = paramValues[id];
  let angle = valueToAngle(value, min, max);
  let dragging = false;

  knob.style.transform = `rotate(${angle}deg)`;
```

- Legge il valore corrente di quel parametro, lo converte in angolo e imposta subito la rotazione, così i knob partono già "allineati".

```
  let startMouseAngle = 0;
```

- Variabile (anche se poi usi `lastMouseAngle`) per tenere traccia dell'angolo del mouse.

```
function mouseAngleDeg(ev, element) {
  const r = element.getBoundingClientRect();
  const cx = r.left + r.width / 2;
  const cy = r.top + r.height / 2;
  const dx = ev.clientX - cx;
  const dy = ev.clientY - cy;
  return Math.atan2(dy, dx) * (180 / Math.PI);
}
```

- Calcola l'angolo rispetto al centro del knob tra centro → puntatore mouse (coordinate viewport), usando `atan2` e convertendo in gradi.

```
knob.addEventListener("mousedown", (e) => {
  e.preventDefault();
  dragging = true;

  lastMouseAngle = mouseAngleDeg(e, knob);

  document.body.style.userSelect = "none";
});
```

- Quando premi sul knob:
  - blocca la selezione di testo;
  - indica che sei in fase di drag;
  - memorizza l'angolo iniziale del mouse;
  - disabilita `userSelect` sul body per evitare selezioni indesiderate mentre trascini.

```
window.addEventListener("mouseup", () => {
  dragging = false;
  document.body.style.userSelect = "";
});
```

- Quando rilasci il mouse ovunque nella finestra, interrompe il drag e ripristina la selezione di testo.

```
window.addEventListener("mousemove", (e) => {
  if (!dragging) return;

  const currentMouseAngle = mouseAngleDeg(e, knob);
  let delta = currentMouseAngle - lastMouseAngle;

  if (delta > 180) delta -= 360;
  if (delta < -180) delta += 360;
```

- Durante il movimento del mouse:
  - se non stai trascinando, esci;
  - calcoli l'angolo attuale;
  - `delta` è la differenza, normalizzata per gestire il passaggio da +180 a -180 (e viceversa) senza salti.

```
  const speed = e.shiftKey ? 0.35 : 1.0;
  angle = clamp(angle + delta * speed, -135, 135);

  lastMouseAngle = currentMouseAngle;
```

- Se `Shift` è premuto, il knob si muove più lentamente (maggiore precisione).

- Aggiorna l'angolo del knob con `delta*speed` e lo clampa entro ±135°.
- Aggiorna `lastMouseAngle` per il passo successivo.

```
  knob.style.transform = `rotate(${angle}deg)`;

  const t = (angle + 135) / 270;
  const raw = lerp(min, max, t);
  const v = Math.round(raw / step) * step;
```

- Ruota graficamente il knob.
- Converte l'angolo (−135/+135) in `t` tra 0 e 1, interpolando il valore numerico tra min e max, poi "snap" al multiplo più vicino dello step.

```
  value = v;
  updateParam(id, v);
  updateValLabel(id, v);
});
```

- Aggiorna la variabile `value`, propaga il nuovo valore nel motore audio (`updateParam`) e aggiorna la label testuale (`updateValLabel`).

```
  updateValLabel(id, value);
  updateParam(id, value);
});
```

- Alla fine del setup del knob, chiama ancora `updateValLabel` e `updateParam` per allineare tutto con il valore iniziale.

Applicare i preset

```
function applyPreset(name) {
  initAudioGraph();

  const preset = presetsConfig[name];
  if (!preset) return;

  const p = preset.params;

  const knobs = document.querySelectorAll(".knob");
```

- Assicura che il grafo audio esista.
- Recupera la definizione del preset; se non c'è, esce.
- `p` è l'oggetto con i parametri da impostare.
- Recupera tutti i knob per poter aggiornare grafica e valori.

```
  knobs.forEach(k => {
    const id = k.dataset.target;
```

```
    if (id === "gain") return;
    if (!(id in p)) return;

    const val = p[id];

    paramValues[id] = val;
    updateParam(id, val);
    updateValLabel(id, val);
```

- Scorre ogni knob:
  - se è il gain, non lo tocca (volume manuale);
  - se l'id non è nel preset, lo salta;
  - altrimenti legge il valore del preset, lo salva in `paramValues`, aggiorna l'audio e la label.

```
    const min = +k.dataset.min;
    const max = +k.dataset.max;
    k.style.transform = `rotate(${valueToAngle(val, min, max)}deg)`;
  });
}
```

- Calcola l'angolo corrispondente al valore del preset e ruota il knob per farselo vedere allineato.

Aggiornare label e parametri audio

```
function updateValLabel(id, v) {
  const el = document.getElementById(id + "Val");
  if (!el) return;
  if (id === "lowpass" || id === "highpass") el.textContent = Math.round(v);
  else el.textContent = v.toFixed(2);
}
```

- Trova la label corrispondente (`gainVal`, `pitchVal` ...) e la aggiorna con formattazione adattata (interi per frequenze, 2 decimali per gli altri).

```
function updateParam(id, v) {
  paramValues[id] = v;

  if (id === "gain" && masterGain) masterGain.gain.value = v / 100;
  else if (id === "lowpass" && lowpassFilter) lowpassFilter.frequency.value = v;
  else if (id === "highpass" && highpassFilter) highpassFilter.frequency.value = v;
  else if (id === "delayTime" && delayNode) delayNode.delayTime.value = v;
  else if (id === "reverbMix" && dryGain && wetGain) {
    wetGain.gain.value = v;
    dryGain.gain.value = 1 - v;
  }
}
```

- Aggiorna la mappa `paramValues`.
- In base all'id, aggiorna il nodo Web Audio corrispondente se esiste (gain scalato da 0–100 a 0–1; frequenze; delay; mix dry/wet).

Disegno waveform

```
function drawWaveform() {
  if (!analyser) return;
  animationId = requestAnimationFrame(drawWaveform);
```

- Se l'`analyser` esiste, pianifica la prossima chiamata a se stessa (loop animato) e salva l'id per diritto di cancellazione.

```
  const w = waveformCanvas.width;
  const h = waveformCanvas.height;

  analyser.getByteTimeDomainData(dataArray);
```

- Legge dimensioni del canvas e riempie `dataArray` con i campioni time-domain normalizzati 0–255.[13]

```
  wfCtx.fillStyle = "#000";
  wfCtx.fillRect(0, 0, w, h);
```

- Pulizia: riempie lo sfondo di nero.

```
  wfCtx.lineWidth = 2;
  wfCtx.strokeStyle = "#38bdf8";
  wfCtx.beginPath();

  const slice = w / dataArray.length;
  let x = 0;
```

- Imposta lo stile della traccia (spessore, colore) e prep l'inizio del path.
- `slice`: distanza orizzontale tra due campioni; `x` parte da 0.

```
  for (let i = 0; i < dataArray.length; i++) {
    const v = dataArray[i] / 128.0;
    const y = v * h / 2;
    if (i === 0) wfCtx.moveTo(x, y);
    else wfCtx.lineTo(x, y);
    x += slice;
  }
```

- Converte ogni byte [0–255] in `v` circa [0–2] e poi lo mappa in coordinate verticali (metà altezza come riferimento).
- Usa `moveTo` per il primo punto, poi `lineTo` per unirli.

```
  wfCtx.lineTo(w, h / 2);
  wfCtx.stroke();
}
```

- Chiude la forma d’onda fino al centro, poi traccia il path.

```
window.addEventListener("beforeunload", () => {
  if (animationId) cancelAnimationFrame(animationId);
});
```

- Alla chiusura/refresh della pagina interrompe il ciclo animato per evitare callback pendenti.

Registrazione microfono (MediaRecorder)

```
btnStartRec.addEventListener("click", async () => {
  try {
    const stream = await navigator.mediaDevices.getUserMedia({ audio: true });
```

- Click su “REC”: chiede il permesso per il microfono e ottiene uno `MediaStream` audio.[16] [17]

```
    mediaRecorder = new MediaRecorder(stream);
    recordedChunks = [];
```

- Crea un `MediaRecorder` che ascolta quello stream e resetta l’array di chunk.[12] [11]

```
    mediaRecorder.ondataavailable = e => {
      if (e.data.size) recordedChunks.push(e.data);
    };
```

- Handler che viene chiamato ogni volta che il `MediaRecorder` ha dati disponibili; si salvano i `Blob` nel vettore se non vuoti.

```
    mediaRecorder.start();
    recStartTime = performance.now();
```

- Avvia la registrazione e memorizza il tempo di inizio (per il timer).

```
    if (recTimerId) clearInterval(recTimerId);
    recTimerId = setInterval(() => {
      const elapsed = (performance.now() - recStartTime) / 1000;
      statusEl.textContent = `Recording... ${elapsed.toFixed(1)}s`;
    }, 100);
```

- (Ri)imposta un intervallo che ogni 100 ms aggiorna lo status con i secondi trascorsi.

```
    btnStartRec.disabled = true;
    btnStopRec.disabled = false;
```

```
    statusEl.textContent = "Recording...";
  } catch (e) {
    console.error(e);
    statusEl.textContent = "Error: the mic is on";
  }
});
```

- Disabilita il bottone start, abilita stop e cambia lo stato.
- In caso di errore (permesso negato, ecc.), logga e mostra un messaggio di errore.

```
btnStopRec.addEventListener("click", () => {
  mediaRecorder.onstop = () => {
    audioBlob = new Blob(recordedChunks, { type: "audio/webm" });
    if (audioUrl) URL.revokeObjectURL(audioUrl);
    audioUrl = URL.createObjectURL(audioBlob);
    player.src = audioUrl;
```

- Click su “STOP”: imposta cosa fare quando la registrazione termina:
  - crea un Blob da tutti i chunk registrati (`audio/webm`);
  - rilascia eventuale vecchio URL;
  - genera una nuova URL di oggetto e la imposta come sorgente del player audio.

```
    btnPlayProcessed.disabled = false;
    btnDownloadWav.disabled = false;
    btnDownloadProcessedWav.disabled = false;
    statusEl.textContent = "The recording is ready";

    refreshDriveButtons();
  };
```

- Abilita i pulsanti Play FX, download WAV grezzo e con effetti, aggiorna lo status, e ricalcola lo stato dei pulsanti Drive.

```
  if (mediaRecorder && mediaRecorder.state === "recording") {
    mediaRecorder.stop();
    mediaRecorder.stream.getTracks().forEach(t => t.stop());
  }
```

- Se c’è un `mediaRecorder` in stato `recording`, lo ferma e ferma tutti i track dello stream (disattiva il microfono).[11] [12]

```
  if (recTimerId) {
    clearInterval(recTimerId);
    recTimerId = null;
  }

  btnStartRec.disabled = false;
```

```
  btnStopRec.disabled = true;
});
```

- Ferma e annulla il timer del display, riabilita il bottone start e disabilita stop.

Eventi del player

```
player.addEventListener("play", () => {
  initAudioGraph();
  connectPlayerToAnalyser();
  if (!animationId) drawWaveform();
```

- Quando il player inizia a suonare:
  - assicura che il grafo audio sia inizializzato;
  - collega il player all'`analyser`;
  - se la waveform non sta già girando, avvia `drawWaveform()`.

```
  if (activeProcessedSource) {
    try {
      activeProcessedSource.stop();
    } catch (e) {}
    activeProcessedSource = null;
  }

  btnPlayProcessed.disabled = false;
});
```

- Se un audio "processato" sta suonando, lo ferma per evitare sovrapposizione.
- Riabilita il tasto Play FX (ad esempio se era stato disabilitato a fine playback FX).

```
player.addEventListener("ended", () => {
  if (animationId) {
    cancelAnimationFrame(animationId);
    animationId = null;
  }
});
```

- Quando il player finisce la traccia, interrompe anche il disegno della waveform.

Download WAV grezzo

```
btnDownloadWav.addEventListener("click", () => {
  downloadWav({getBlobFn: getRawWavBlob, prefix: "Aurora_"});
});
```

- Al click, chiama `downloadWav`, passando una funzione che restituirà il Blob WAV grezzo e un prefisso per il nome file.

```
async function downloadWav({getBlobFn, prefix}) {
  if (!audioBlob) return;
```

- Se non c'è una registrazione, esce.

```
  try {
    const filename = safeTimestampName("wav", prefix);
    const wavBlob = await getBlobFn();
    downloadBlob(wavBlob, filename);
    statusEl.textContent = `Downloaded - ${filename}`;
  } catch (e) {
    console.error(e);
    statusEl.textContent = "Error while exporting the file";
  }
}
```

- Genera un nome file con timestamp, ottiene il Blob WAV (via `getBlobFn`), lo scarica, e aggiorna il messaggio di stato; in caso di errori mostra un messaggio.

```
function safeTimestampName(ext, prefix) {
  const d = new Date();
  const pad = (n) => String(n).padStart(2, "0");
  const stamp = `${d.getFullYear()}-${pad(d.getMonth()+1)}-${pad(d.getDate())}_${pad(d.ge
  return `${prefix}${stamp}.${ext}`;
}
```

- Crea una stringa timestamp `YYYY-MM-DD_HH-MM-SS` e la concatena a prefisso + estensione.

```
function downloadBlob(blob, filename) {
  const url = URL.createObjectURL(blob);
  const a = document.createElement("a");
  a.style.display = "none";
  a.href = url;
  a.download = filename;
  document.body.appendChild(a);
  a.click();
  URL.revokeObjectURL(url);
  document.body.removeChild(a);
}
```

- Crea un link invisibile con `href` alla URL blob, imposta `download`, lo clicca programmaticamente per lanciare il download, poi pulisce URL e DOM.

```
async function getRawWavBlob() {
  if (!audioBlob) throw new Error("No recording available");
  initAudioGraph();
  const arrayBuffer = await audioBlob.arrayBuffer();
  const audioBuffer = await audioCtx.decodeAudioData(arrayBuffer);
```

```
  const wavBuffer = audioBufferToWav(audioBuffer);
  return new Blob([wavBuffer], { type: "audio/wav" });
}
```

- Verifica che esista una registrazione.
- Inizializza contesto audio.
- Converte `audioBlob` in `ArrayBuffer`, poi in `AudioBuffer` con `decodeAudioData`.
- Usa `audioBufferToWav` (libreria esterna) per ottenere un buffer WAV e costruisce un Blob `audio/wav` pronto per il download.[18] [13]

Playback con effetti

```
btnPlayProcessed.addEventListener("click", async () => {
  if (!audioBlob) return;
  initAudioGraph();
```

- Se non c'è audio, esce; altrimenti assicura che la catena audio sia pronta.

```
  if (!player.paused) {
    player.pause();
    player.currentTime = 0;
  }
```

- Se il player sta suonando, lo ferma e riporta a inizio per non avere due playback in parallelo.

```
  if (activeProcessedSource) {
    try {
      activeProcessedSource.stop();
    } catch (e) {}
    activeProcessedSource = null;
  }
```

- Se c'era già un playback FX in corso, lo ferma e azzera il riferimento.

```
  const arrayBuffer = await audioBlob.arrayBuffer();
  const audioBuffer = await audioCtx.decodeAudioData(arrayBuffer);
  const source = audioCtx.createBufferSource();
  source.buffer = audioBuffer;
  source.playbackRate.value = paramValues.pitch;
  source.connect(lowpassFilter);
```

- Decodifica il Blob in `AudioBuffer`, crea un `BufferSource`, assegna il buffer, imposta il playback rate per simulare pitch, e collega la sorgente all'inizio della catena (lowpass).[13]

```
  activeProcessedSource = source;
  btnPlayProcessed.disabled = true;
```

- Salva la sorgente come “attiva” e disabilita il pulsante finché non finisce.

```
  source.onended = () => {
    if (activeProcessedSource === source) {
      activeProcessedSource = null;
    }
    btnPlayProcessed.disabled = false;
  };
```

- Quando il playback termina, se è la sorgente ancora attiva, azzera il riferimento e riabilita il pulsante Play FX.

```
  source.start();
  if (!animationId) drawWaveform();
});
```

- Avvia la riproduzione del segnale processato e, se la waveform non è in esecuzione, la avvia.

Download WAV con effetti (render offline)

```
btnDownloadProcessedWav.addEventListener("click", () => {
  downloadWav({getBlobFn: getProcessedWavBlob, prefix: "Aurora_fx_"});
});
```

- Usa lo stesso meccanismo di `downloadWav`, ma ottenendo un Blob che include gli effetti.

```
async function getProcessedWavBlob() {
  if (!audioBlob) throw new Error("No recording available");
```

- Verifica che esista una registrazione.

```
  const arr = await audioBlob.arrayBuffer();
  const probeCtx = new (window.AudioContext || window.webkitAudioContext)();
  const decoded = await probeCtx.decodeAudioData(arr);
  const duration = decoded.duration;
  const sampleRate = decoded.sampleRate;
  probeCtx.close();
```

- Usa un `AudioContext` “di appoggio” per decodificare il Blob e ricavare durata e sample rate del segnale (poi chiude immediatamente il context).[13]

```
  const length = Math.ceil(duration * sampleRate);
  const offlineCtx = new (window.OfflineAudioContext || window.webkitOfflineAudioContext)
```

- Calcola il numero di campioni richiesti per rappresentare l’intero file e crea un `OfflineAudioContext` mono con quella lunghezza e sample rate.[19] [13]

```
const source = offlineCtx.createBufferSource();
source.buffer = decoded;
```

- Crea una sorgente buffer offline e le assegna il buffer decodificato.

Poi ricostruisce la stessa catena di effetti, ma in versione offline:

```
const lp = offlineCtx.createBiquadFilter();
lp.type = "lowpass";
lp.frequency.value = paramValues.lowpass;

const hp = offlineCtx.createBiquadFilter();
hp.type = "highpass";
hp.frequency.value = paramValues.highpass;

const del = offlineCtx.createDelay(5.0);
del.delayTime.value = paramValues.delayTime;

const fb = offlineCtx.createGain();
fb.gain.value = 0.3;
del.connect(fb);
fb.connect(del);

const conv = offlineCtx.createConvolver();
conv.buffer = createReverbImpulse(offlineCtx, 2.5, 2.0);

const dry = offlineCtx.createGain();
const wet = offlineCtx.createGain();
const master = offlineCtx.createGain();

master.gain.value = paramValues.gain;
const mix = paramValues.reverbMix;
dry.gain.value = 1 - mix;
wet.gain.value = mix;

source.playbackRate.value = paramValues.pitch;
```

- Stesse logiche di `initAudioGraph`, ma tutto dentro l'OfflineAudioContext.[19] [13]

```
source.connect(lp);
lp.connect(hp);
hp.connect(del);
del.connect(dry);
del.connect(conv);
conv.connect(wet);
dry.connect(master);
wet.connect(master);
master.connect(offlineCtx.destination);
```

- Connetti la catena: lowpass → highpass → delay → (dry + wet) → master → destinazione offline (buffer renderizzato).

```
  source.start(0);
  const rendered = await offlineCtx.startRendering();
```

- Fa partire la sorgente e chiede all'OfflineAudioContext di renderizzare il grafo. Il risultato è un `AudioBuffer` con l'audio già "effettato".[19]

```
  const wavBuffer = audioBufferToWav(rendered);
  return new Blob([wavBuffer], { type: "audio/wav" });
}
```

- Converte il buffer in WAV e ritorna un Blob `audio/wav` pronto per upload/download.[18]

Google Drive – inizializzazione OAuth

```
function initDriveAuth() {
  if (!btnAuthDrive || !window.google || !google.accounts || !google.accounts.oauth2) {
    return;
  }
  if (tokenClient) return;
```

- Se il pulsante non c'è o la libreria Google non è stata ancora caricata, non fa nulla.
- Se `tokenClient` è già stato creato, non reinizializza.

```
  if (!GOOGLE_CLIENT_ID || GOOGLE_CLIENT_ID.includes("PASTE_YOUR_CLIENT_ID_HERE")) {
    setDriveStatus("Drive: insert GOOGLE_CLIENT_ID in script.js");
    btnAuthDrive.disabled = true;
    return;
  }
```

- Se il CLIENT ID non è impostato correttamente, mostra un warning e disabilita il pulsante.

```
  btnAuthDrive.addEventListener("click", () => {
    setDriveStatus("Drive: authorization in progress...");
    tokenClient.requestAccessToken({ prompt: "consent" });
  });
```

- Quando l'utente clicca su "Google Drive Authorization", aggiorna lo stato e chiama `requestAccessToken`, che apre il flusso di login e consenso.[14]

```
  tokenClient = google.accounts.oauth2.initTokenClient({
    client_id: GOOGLE_CLIENT_ID, scope: DRIVE_SCOPES,
    callback: (resp) => {
      driveAccessToken = resp.access_token;
      driveFolderIdCache = null;
      setDriveStatus("Drive: authorized");
      refreshDriveButtons();
    },
```

```
  });
}
```

- Inizializza `tokenClient` con client ID, scope e una callback che viene chiamata alla fine del flusso OAuth: salva l'access token, resetta la cache cartella, aggiorna lo stato e abilita i pulsanti upload.[15] [14]

```
window.addEventListener("load", () => {
  initDriveAuth();
  let tries = 0;
  const t = setInterval(() => {
    initDriveAuth();
    tries++;
    if (tokenClient || tries > 40) clearInterval(t);
  }, 250);
});
```

- Alla fine del caricamento pagina prova subito a inizializzare l'OAuth.
- Usa anche un `setInterval` per ritentare ogni 250 ms finché `tokenClient` non è stato creato o si raggiungono 40 tentativi (circa 10s), per coprire eventuali ritardi nel caricamento dello script Google.

Stato e pulsanti Drive

```
function setDriveStatus(msg) {
  if (driveStatusEl) driveStatusEl.textContent = msg;
}
```

- Aggiorna il testo netto nello span sotto i pulsanti Drive.

```
function refreshDriveButtons() {
  const ok = canUploadNow();
  if (btnUploadWav) btnUploadWav.disabled = !ok;
  if (btnUploadProcessedWav) btnUploadProcessedWav.disabled = !ok;
}
```

- Abilita i pulsanti upload solo se la condizione `canUploadNow()` è vera.

```
function canUploadNow() {
  return !!driveAccessToken && !!audioBlob;
}
```

- Restituisce `true` solo se hai un token di accesso valido e una registrazione pronta (Blob).

Upload Google Drive

```
if (btnUploadWav) {
  btnUploadWav.addEventListener("click", async () => {
    try {
      setDriveStatus("Drive: upload in progress...");
      const folderId = await getOrCreateAuroraFolderId();
      const wavBlob = await getRawWavBlob();
      const filename = safeTimestampName("wav", "Aurora_");
      setDriveStatus("Drive: loading WAV...");
      const fileId = await uploadBlobToDriveResumable(wavBlob, filename, "audio/wav", fol
      setDriveStatus(`Drive: uploaded - ${filename}`);
      console.log("Drive fileId (mic):", fileId);
    } catch (e) {
      console.error(e);
      setDriveStatus("Drive: upload failed");
    }
  });
}
```

- Click “Upload WAV to Drive”:
  - Aggiorna stato;
  - ottiene l’id cartella (creandola se necessario);
  - genera Blob WAV grezzo e nome file;
  - chiama upload resumable;
  - aggiorna stato con “uploaded” e logga l’id;
  - in caso di errore, mostra “upload failed”.[20]

```
if (btnUploadProcessedWav) {
  btnUploadProcessedWav.addEventListener("click", async () => {
    try {
      setDriveStatus("Drive: upload in progress...");
      const folderId = await getOrCreateAuroraFolderId();
      const wavBlob = await getProcessedWavBlob();
      const filename = safeTimestampName("wav", "Aurora_fx_");
      setDriveStatus("Drive: loading WAV with effects...");
      const fileId = await uploadBlobToDriveResumable(wavBlob, filename, "audio/wav", fol
      setDriveStatus(`Drive: uploaded - ${filename}`);
      console.log("Drive fileId (fx):", fileId);
    } catch (e) {
      console.error(e);
      setDriveStatus("Drive: upload failed");
    }
  });
}
```

- Stesso identico flusso, ma usando il Blob con effetti.

Upload resumable e gestione cartella

```
async function uploadBlobToDriveResumable(blob, filename, mimeType, folderId) {
  const metadata = {
    name: filename,
    parents: [folderId],
  };
```

- Prepara un oggetto JSON con nome file e cartella di destinazione.

```
  const start = await driveFetch("https://www.googleapis.com/upload/drive/v3/files?upload
    method: "POST",
    headers: {
      "Content-Type": "application/json; charset=UTF-8",
      "X-Upload-Content-Type": mimeType,
      "X-Upload-Content-Length": String(blob.size),
    },
    body: JSON.stringify(metadata),
  });
```

- Prima fase del resumable upload: `POST` all'endpoint Drive upload con `uploadType=resumable`, header che descrivono il contenuto e body con metadati.[20]

```
  if (!start.ok) throw new Error(await start.text());
  const uploadUrl = start.headers.get("Location");
  if (!uploadUrl) throw new Error("Missing upload URL");
```

- Se la risposta non è ok, solleva errore; altrimenti legge la URL di upload dalla header "Location".

```
  const put = await fetch(uploadUrl, {
    method: "PUT",
    headers: {
      "Content-Type": mimeType,
      "Content-Length": String(blob.size),
    },
    body: blob,
  });
```

- Seconda fase: `PUT` diretto del contenuto (blob MP3/WAV) all'URL di sessione.

```
  if (!put.ok) throw new Error(await put.text());
  const data = await put.json();
  return data.id;
}
```

- Se ok, parse del JSON e ritorna l'id file: se ti serve, puoi salvarlo da qualche parte.

```
async function getOrCreateAuroraFolderId() {
```

```
  if (driveFolderIdCache) return driveFolderIdCache;
```

- Se abbiamo già la cartella in cache, ritorniamo quell'id subito.

```
  const q = `name='${DRIVE_FOLDER_NAME.replace(/'/g, "\\'")}' and mimeType='application/v
  const listUrl = `https://www.googleapis.com/drive/v3/files?q=${encodeURIComponent(q)}&f
```

- Costruisce una query Drive: nome uguale alla costante, tipo "folder" e non cestinata, codificata per l'URL.[20]

```
  const r = await driveFetch(listUrl);
  if (!r.ok) throw new Error(await r.text());
  const data = await r.json();
  if (data.files && data.files.length) {
    driveFolderIdCache = data.files[^2_0].id;
    return driveFolderIdCache;
  }
```

- Chiede la lista delle cartelle che soddisfano la query; se ne trova almeno una, usa la prima e la mette in cache.

```
  const createUrl = "https://www.googleapis.com/drive/v3/files";
  const body = {
    name: DRIVE_FOLDER_NAME,
    mimeType: "application/vnd.google-apps.folder",
  };
  const c = await driveFetch(createUrl, {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify(body),
  });
```

- Se la cartella non esiste, prepara una richiesta `POST` all'endpoint Drive files per crearla, con body JSON di tipo "folder".[20]

```
  if (!c.ok) throw new Error(await c.text());
  const created = await c.json();
  driveFolderIdCache = created.id;
  return driveFolderIdCache;
}
```

- Se la creazione va a buon fine, legge il JSON di risposta, memorizza l'id e lo ritorna.

```
async function driveFetch(url, options = {}) {
  if (!driveAccessToken) throw new Error("Drive not authorized");
  const headers = new Headers(options.headers || {});
  headers.set("Authorization", `Bearer ${driveAccessToken}`);
  return fetch(url, { ...options, headers });
}
```

- Wrapper di `fetch` che:
  - controlla che esista un access token (altrimenti errore);
  - aggiunge header `Authorization: Bearer <token>` a tutte le richieste verso le API di Drive;
  - chiama `fetch` con le opzioni originali + header di autenticazione.[14]

Se vuoi, posso aiutarti a estrarre alcune di queste parti in moduli separati (es. un file “audio-engine.js” e uno “drive.js”) per rendere il progetto più manutenibile.

⁂

1. https://www.npmjs.com/package/audiobuffer-to-wav?activeTab=versions
2. https://web.dev/patterns/media/microphone-record
3. https://forum.duplicacy.com/t/reducing-google-drive-scope-a-simple-proposal/3625
4. https://stackoverflow.com/questions/51274573/how-can-i-use-the-drive-file-oauth2-scope-with-a-folder-in-google-drive
5. https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia
6. https://24ways.org/2019/building-a-dictaphone-using-media-recorder-and-getusermedia/
7. https://addpipe.com/getusermedia-examples/
8. https://www.npmjs.com/package/audiobuffer-loader
9. https://www.antopiras.dev/blog/2021-05-15-using-the-mediastream-web-api-to-record-screen-camera-and-audio/
10. https://stackoverflow.com/questions/69669177/how-do-i-use-the-mediarecorder-api-on-processed-audio
11. https://developer.mozilla.org/nl/docs/Web/API/MediaRecorder
12. https://developer.mozilla.org/en-US/docs/Web/API/MediaStream_Recording_API/Using_the_MediaStream_Recording_API
13. https://mdn.org.cn/en-US/docs/Web/API/OfflineAudioContext
14. https://developers.google.com/identity/oauth2/web/guides/use-token-model
15. https://developers.google.com/identity/oauth2/web/reference/js-reference
16. https://developer.mozilla.org/en-US/docs/Web/API/MediaDevices/getUserMedia
17. https://www.devdoc.net/web/developer.mozilla.org/en-US/docs/Web/API/Navigator/mediaDevices/getUserMedia.html
18. https://www.npmjs.com/package/audiobuffer-to-wav?activeTab=versions
19. https://mdn2.netlify.app/en-us/docs/web/api/offlineaudiocontext/startrendering/
20. https://stackoverflow.com/questions/51274573/how-can-i-use-the-drive-file-oauth2-scope-with-a-folder-in-google-drive
21. https://24ways.org/2019/building-a-dictaphone-using-media-recorder-and-getusermedia/
22. https://stackoverflow.com/questions/50095997/getusermedia-recording-do-not-stop-after-wait-on-mdn-example