// --------------------------- INSTRUCTIONS (copy+share to AI bot) ---------------------------

// This single file contains the minimal runtime helpers, TTS and STT client changes needed to

// make TTS work reliably on macOS/iOS Safari and to avoid the client POSTing to localhost.

// Steps to apply after restoring your backup:

// 1) Paste the code blocks below into your existing flutter\_client/lib/main.dart file,

// replacing/merging with the existing sections as indicated by "...existing code..." markers.

// 2) Ensure installRuntimeHelpers() is called early in startup (main() or State.initState()).

// Example: void main(){ installRuntimeHelpers(); runApp(MyApp()); }

// 3) Rebuild Flutter web: flutter build web

// 4) Deploy: copy flutter\_client/build/web/\* -> server/public/ (overwrite), then restart the Node server

// or deploy the server to Cloud Run. For quick external testing you may run an HTTPS tunnel (ngrok/localtunnel).

// 5) IMPORTANT: On test devices unregister stale service workers and hard-refresh.

// In browser console run:

// navigator.serviceWorker.getRegistrations().then(r=>r.forEach(x=>x.unregister()));

// Also press the in-app "Enable sound + mic" button (or tap the page) once on iOS to unlock audio.

// Verification:

// - TTS: use app UI that calls useDirectTTS(text). The client will fetch /api/tts from a runtime-resolved base,

// create an AudioElement, attempt play, and queue audio if playback is blocked until a user gesture.

// - STT: convertSpeechToText() builds upload URL from resolveServerBase() and uses fetch via JS interop.

// -----------------------------------------------------------------------------------------

// ...existing code...

import 'dart:async';

import 'dart:convert';

import 'dart:html' as html;

import 'dart:js' as js;

import 'dart:js\_util' as js\_util;

import 'package:flutter/material.dart';

import 'package:flutter/services.dart';

import 'package:http/http.dart' as http;

import 'package:provider/provider.dart';

import 'providers/persona\_provider.dart';

import 'providers/settings\_provider.dart';

// ...existing code...

// --- BEGIN PATCH: runtime resolver, helpers, TTS + STT snippets ---

// Compile-time default (keep as-is)

const String SERVER\_BASE = String.fromEnvironment('SERVER\_BASE', defaultValue: 'http://localhost:3000');

// Resolve server base at runtime: prefer window.SERVER\_BASE, infer/upgrade on HTTPS pages

String resolveServerBase() {

try {

final jsBase = js\_util.getProperty(js.context, 'SERVER\_BASE');

if (jsBase is String && jsBase.isNotEmpty) {

try {

final loc = html.window.location;

if (loc.protocol == 'https:' && jsBase.contains('localhost')) {

if (loc.host.contains('-web-')) {

final apiHost = loc.host.replaceAll('-web-', '-api-');

final inferred = '${loc.protocol}//$apiHost';

try { js\_util.setProperty(js.context, 'SERVER\_BASE', inferred); } catch (\_) {}

return inferred;

}

if (jsBase.startsWith('http://')) return 'https://' + jsBase.substring('http://'.length);

}

} catch (\_) {}

return jsBase;

}

} catch (\_) {}

var base = SERVER\_BASE;

try {

final loc = html.window.location;

if (loc.protocol == 'https:' && base.startsWith('http://')) {

if (base.contains('localhost') && loc.host.contains('-web-')) {

final apiHost = loc.host.replaceAll('-web-', '-api-');

final inferred = '${loc.protocol}//$apiHost';

return inferred;

}

base = 'https://' + base.substring('http://'.length);

}

} catch (\_) {}

return base;

}

// Install JS runtime helpers: expose SERVER\_BASE, unregister SWs, audio-unlock helpers

void installRuntimeHelpers() {

try { js.context['SERVER\_BASE'] = SERVER\_BASE; } catch (\_) {}

try {

js.context.callMethod('eval', [r'''

try {

if (window.SERVER\_BASE && window.SERVER\_BASE.indexOf('localhost') !== -1 && location.protocol === 'https:') {

var host = location.hostname || '';

var apiHost = host;

if (host.indexOf('-web-') !== -1) apiHost = host.replace('-web-','-api-');

else if (host.indexOf('web') !== -1) apiHost = host.replace('web','api');

window.SERVER\_BASE = 'https://' + apiHost;

console.log('[net] Resolved SERVER\_BASE at runtime ->', window.SERVER\_BASE);

if (window.dartAppendTranscript) window.dartAppendTranscript('[net] Resolved SERVER\_BASE at runtime -> ' + window.SERVER\_BASE);

} else {

console.log('[net] Using SERVER\_BASE ->', window.SERVER\_BASE);

}

} catch(e) { console.error('SERVER\_BASE runtime resolver error', e); }

''']);

} catch (\_) {}

// Unregister service workers (avoid stale cache of old builds)

try {

js.context.callMethod('eval', [r'''

(function(){

if ('serviceWorker' in navigator) {

navigator.serviceWorker.getRegistrations().then(function(regs){

regs.forEach(function(r){ r.unregister().catch(function(){}); });

console.log('[net] Unregistered existing service workers');

}).catch(function(){});

}

})();

''']);

} catch (\_) {}

// resolveRuntimeServerBase + audio-unlock + pending audio queue helpers

try {

js.context.callMethod('eval', [r'''

(function(){

try {

if (!window.resolveRuntimeServerBase) {

window.resolveRuntimeServerBase = function(){

try {

var b = window.SERVER\_BASE || '';

var loc = window.location || { protocol: 'https:', host: '', origin: '' };

if (loc.protocol === 'https:') {

if (b && b.indexOf('localhost') !== -1 && loc.host.indexOf('-web-') !== -1) {

return loc.protocol + '//' + loc.host.replace('-web-', '-api-');

}

if (b && b.indexOf('http://') === 0) {

return 'https://' + b.substring('http://'.length);

}

}

if (b) return b;

return loc.origin || (loc.protocol + '//' + loc.host);

} catch(e) { return ''; }

};

console.log('[net] resolveRuntimeServerBase installed ->', window.resolveRuntimeServerBase());

}

} catch(e){}

})();

''']);

} catch (\_) {}

try {

js.context.callMethod('eval', [r'''

(function(){

if (window.\_\_audioUnlockInstalled) return;

window.\_\_audioUnlockInstalled = true;

window.\_\_pendingAudioElements = [];

window.\_\_audioUnlocked = false;

function replayPending(){

try {

if (Array.isArray(window.\_\_pendingAudioElements)) {

window.\_\_pendingAudioElements.forEach(function(a){ try{ a.play().catch(()=>{});}catch(e){} });

window.\_\_pendingAudioElements.length = 0;

}

} catch(e) {}

}

function unlock(){

try {

if (!window.audioContext) {

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

}

if (window.audioContext && window.audioContext.state !== 'running') {

window.audioContext.resume().then(function(){ window.\_\_audioUnlocked = true; replayPending(); }).catch(()=>{});

}

var buf = window.audioContext.createBuffer(1,1,22050);

var src = window.audioContext.createBufferSource();

src.buffer = buf;

src.connect(window.audioContext.destination);

try { src.start(0); } catch(e){}

window.\_\_audioUnlocked = true;

replayPending();

} catch(e) {}

}

window.\_\_forceAudioUnlock = function(){ unlock(); };

window.addEventListener('touchend', unlock, { once: true, capture: true });

window.addEventListener('click', unlock, { once: true, capture: true });

})();

''']);

} catch (\_) {}

}

// Example: replace your existing direct TTS call with this implementation.

// It uses resolveServerBase() and queues audio if playback is blocked.

Future<void> useDirectTTS(String text, {String? voice}) async {

try {

appendTranscript('[info] Using direct TTS API: $text');

String resolvedBase = resolveServerBase();

try {

final jsBase = js\_util.getProperty(html.window, 'SERVER\_BASE');

if (jsBase is String && jsBase.isNotEmpty) resolvedBase = jsBase;

} catch (\_) {}

try {

if (html.window.location.protocol == 'https:' && resolvedBase.startsWith('http://')) {

resolvedBase = resolvedBase.replaceFirst('http://', 'https://');

}

} catch (\_) {}

final uri = Uri.parse('${resolvedBase.replaceAll(RegExp(r'/$'), '')}/api/tts');

final response = await http.post(

uri,

headers: {'Content-Type': 'application/json'},

body: json.encode({'text': text, if (voice != null) 'voice': voice}),

);

if (response.statusCode == 200) {

appendTranscript('[info] TTS API response received, preparing playback...');

final bytes = response.bodyBytes;

final audioBlob = html.Blob([bytes], 'audio/mpeg');

final audioUrl = html.Url.createObjectUrlFromBlob(audioBlob);

final audioElement = html.AudioElement()

..src = audioUrl

..autoplay = true

..muted = false

..controls = false

..preload = 'auto';

audioElement.setAttribute('playsinline', 'true');

audioElement.style.display = 'none';

html.document.body?.append(audioElement);

try { await audioElement.play(); } catch (\_) {}

if (audioElement.paused) {

try {

final pending = js\_util.getProperty(js.context, '\_\_pendingAudioElements');

if (pending != null) { js\_util.callMethod(pending, 'push', [audioElement]); }

} catch(\_) {}

appendTranscript('[info] Tap Enable sound to allow playback');

return;

}

appendTranscript('[success] Direct TTS audio playing');

} else {

appendTranscript('[error] Failed to get TTS: ${response.statusCode}');

print('TTS API error: ${response.body}');

}

} catch (e) {

appendTranscript('[error] Exception using TTS API: $e');

print('Error with TTS API: $e');

}

}

// STT: use runtime-resolved base when building the upload URL; example snippet to integrate

Future<void> convertSpeechToText(/\* params \*/) async {

// ...existing code...

String apiBase = resolveServerBase();

try {

final jsBase = js\_util.getProperty(html.window, 'SERVER\_BASE');

if (jsBase is String && jsBase.isNotEmpty) apiBase = jsBase;

} catch (\_) {}

try {

if (html.window.location.protocol == 'https:' && apiBase.startsWith('http://')) {

apiBase = apiBase.replaceFirst('http://', 'https://');

}

} catch (\_) {}

try {

if (apiBase.contains('localhost')) {

final loc = html.window.location;

if (loc.protocol == 'https:' && loc.host.contains('-web-')) {

final inferred = '${loc.protocol}//${loc.host.replaceAll('-web-', '-api-')}';

apiBase = inferred;

try { js\_util.setProperty(js.context, 'SERVER\_BASE', inferred); } catch (\_) {}

appendTranscript('[diag] Inferred API base -> ' + apiBase);

}

}

} catch (\_) {}

final apiUrl = '${apiBase.replaceAll(RegExp(r'/$'), '')}/api/stt?ts=${DateTime.now().millisecondsSinceEpoch}';

appendTranscript('[diag] STT POST -> ' + apiUrl);

try {

final jsFetch = js\_util.getProperty(js.context, 'fetch');

if (jsFetch == null) {

appendTranscript('[error] No fetch() available in this environment');

// ...existing cleanup/state updates...

return;

}

// Build FormData and perform fetch via js interop (adapt to your code's blob/form)

// Example:

// final formData = js\_util.newObject(); // create FormData via interop or use js\_util.callMethod(...)

// final fetchOpts = js\_util.jsify({'method':'POST','mode':'cors','body':formData});

// final fetchPromise = js\_util.callMethod(js.context, 'fetch', [apiUrl, fetchOpts]);

// final resp = await js\_util.promiseToFuture(fetchPromise);

// ...handle resp...

} catch (e) {

appendTranscript('[error] STT upload exception: $e');

// ...existing cleanup/state updates...

}

// ...existing code...

}

// --- END PATCH ---

// ...existing code...

// Reminder: call installRuntimeHelpers() at startup prior to UI interactions.

// Example: in main() or initState().

// ...existing code...