1. MainProgram – Chatbot Tenki-Chan

1.1. Gambaran Umum Alur Chatbot

`MainProgram.cs` adalah skrip Unity (turunan `MonoBehaviour`) yang mengelola seluruh alur percakapan dengan Tenki-Chan. Berikut ringkasan alur kerja saat pengguna berinteraksi dengan chatbot:

1. **Inisialisasi:** Saat aplikasi mulai, *MainProgram* melakukan inisialisasi layanan Unity dan mengambil kunci API (OpenAI, WeatherAPI, ElevenLabs) melalui Unity Cloud Code (opsional). Objek UI (input field, tombol kirim, teks output, dsb.) terhubung di *Inspector* Unity.
2. **Pengiriman Pesan Pengguna:** Ketika pengguna mengetik pesan dan menekan tombol Kirim, fungsi `OnSendClicked()` dipanggil. Fungsi ini menyiapkan pesan untuk diproses: memvalidasi input dan kunci API, menonaktifkan input sementara, serta memulai *coroutine* utama `ProcessUserMessage()` untuk memproses permintaan tersebut.
3. **Meminta Rencana ke OpenAI:** *Coroutine* `ProcessUserMessage` pertama-tama mengirim prompt ke OpenAI Chat Completions untuk mendapatkan **TenkiPlan** dalam format JSON. *Plan* ini mencakup intent pengguna (`"weather"` atau `"chitchat"`), serta detail lain seperti lokasi, waktu, parameter API cuaca, dan jawaban obrolan jika intent adalah chitchat.
4. **Pemrosesan Berdasarkan Intent:**
5. Jika intent = `"chitchat"`, sistem langsung mengambil `plan.reply` (balasan obrolan) dari hasil OpenAI dan menampilkannya.
6. Jika intent = `"weather"`, sistem akan:
    - Menentukan lokasi query (misal koordinat latitude, longitude atau nama tempat) dari *plan* atau teks pengguna.
    - Jika perlu, memperbaiki query lokasi menggunakan fitur *search* WeatherAPI (mencari koordinat dari nama tempat).
    - Menentukan apakah butuh cuaca saat ini (`current`) atau prakiraan (`forecast`) berdasarkan waktu yang diminta (misal, jika pengguna tanya cuaca besok, pakai forecast).
    - Mengambil data cuaca dari WeatherAPI (endpoint *current* atau *forecast*).
    - Mengonversi data cuaca mentah menjadi teks jawaban yang *natural* dan informatif menggunakan sekali lagi OpenAI (dengan prompt khusus format laporan cuaca).
7. **Output dan TTS:** Jawaban akhir (baik dari obrolan biasa maupun laporan cuaca) ditampilkan ke layar. Kemudian, *MainProgram* memulai *coroutine* TTS untuk mendapatkan audio dari ElevenLabs dan memutarnya kepada pengguna. Setelah itu, input diaktifkan kembali sehingga pengguna dapat melanjutkan percakapan.

Berikut ilustrasi potongan kode `OnSendClicked()` yang menangani ketika pengguna menekan tombol Kirim:

```
public void OnSendClicked()
{
    statusText.text = "Sedang Memulai";
    // Pastikan kunci API tersedia (jika tidak menggunakan relay)
    if (string.IsNullOrWhiteSpace(OpenAIApiKey) && !UseSecureRelay)
    {
        SafeOutput("⚠ Error dengan API key LLM. Mohon hubungi developer.");
```

```
            return;
        }
        if (string.IsNullOrWhiteSpace(WeatherApiKey) && !UseSecureRelay)
        {
            SafeOutput("⚠ Error dengan WeatherAPI key. Mohon hubungi developer.");
            return;
        }
        // Ambil teks input pengguna
        var text = InputField != null ? InputField.text.Trim() : "";
        if (string.IsNullOrEmpty(text))
        {
            SafeOutput("Ketik sesuatu untuk Tenki-Chan dahulu ☺");
            return;
        }
        // Hentikan operasi sebelumnya jika masih berjalan
        _epoch++;
        if (_pipeline != null) StopCoroutine(_pipeline);
        CancelInFlight();       // batalkan request jaringan yang masih berjalan
        audioSource.Stop();     // hentikan audio TTS jika ada
        // Mulai coroutine utama pemrosesan pesan
        _pipeline = StartCoroutine(ProcessUserMessage(text, _epoch));
    }
```

Penjelasan:
- Kode di atas memastikan kunci API tersedia (kecuali jika menggunakan server *relay* untuk menyembunyikan kunci). Jika tidak ada, akan menampilkan pesan error lewat `SafeOutput`.
- Jika input kosong, akan diminta mengetik dahulu.
- Lalu, mekanisme `_epoch` digunakan untuk melacak sesi percakapan terbaru. Setiap kali pengguna mengirim pesan, `epoch` bertambah dan *coroutine* lama (jika ada) dihentikan. Hal ini untuk mencegah balasan *terlambat* dari permintaan sebelumnya mempengaruhi antarmuka (misalnya, jika user cepat mengetik pertanyaan baru sebelum jawaban sebelumnya selesai). `CancelInFlight()` akan melakukan **abort** pada semua request network yang masih berlangsung untuk sesi sebelumnya.
- Terakhir, *coroutine* `ProcessUserMessage` dimulai untuk memproses input pengguna terbaru. Sementara proses berlangsung, tombol dan kolom input dinonaktifkan (lihat `SetInteractable(false)` di awal coroutine) untuk mencegah interaksi ganda.

1.2. Mendapatkan *Plan* dari OpenAI (Intent dan Detail)

Setelah `OnSendClicked` memulai *coroutine* `ProcessUserMessage`, langkah pertama adalah meminta *model* OpenAI untuk menentukan maksud (intent) pengguna dan bagaimana menanganinya. Fungsi coroutine `GetTenkiPlanFromLLM` akan menyiapkan *prompt* sistem dan pengguna, mengirim request ke endpoint OpenAI Chat Completions, dan mengurai hasil JSON menjadi objek `TenkiPlan`.

Mempersiapkan System Prompt:
`MainProgram` membangun pesan *system* yang **sangat ketat** menginstruksikan model untuk **hanya** merespons dengan JSON sesuai skema yang telah ditentukan. Ini penting agar output mudah di-*parse* dan

tidak ada teks bebas yang tidak terstruktur. Pembangunan prompt ini dilakukan di fungsi `BuildSystemPrompt()`. Berikut adalah beberapa aturan yang terdapat dalam *system prompt* tersebut (ringkas dalam Bahasa Indonesia):

- Model **hanya boleh output** sebuah objek JSON tunggal sesuai skema (tidak boleh ada teks lain, tidak ada markdown, tidak ada penjelasan).
- **Skema JSON** mencakup:
- `intent`: `"weather"` atau `"chitchat"`.
- `reply`: string balasan jika intent chitchat (Bahasa Indonesia saja).
- `location`: objek berisi `query` (nama lokasi), `lat`, `lon`, `country`, `admin` (provinsi/negara bagian) – boleh kosong atau parsial jika model tidak yakin.
- `time`: objek waktu dengan `type` (`"now"` / `"date"` / `"relative"`), serta rincian tanggal, waktu, zona waktu jika relevan.
- `weather_api`: objek parameter untuk WeatherAPI – `endpoint` (`"current"` / `"forecast"`), `q` (query lokasi, bisa "lat,lon" atau nama), `days` (jumlah hari prakiraan), `dt` (tanggal khusus), `units` ("metric"/"imperial").
- Terdapat aturan tambahan, misalnya: jika user minta cuaca saat ini, gunakan `time.type = now` dan `weather_api.endpoint = current`. Jika menanyakan tanggal tertentu atau hari esok/lusa, gunakan `forecast`. Untuk lokasi di Indonesia, diusahakan memberikan lat/lon jika mungkin, atau gunakan query nama tempat yang spesifik.

Prompt pengguna yang dikirim ke model mencakup teks asli pengguna serta *previousUserText* (yaitu prompt pengguna sebelumnya, untuk konteks jika diperlukan). Dengan menyediakan percakapan sebelumnya, model bisa memahami konteks, namun di sini yang diberikan hanya prompt terakhir user dan prompt sebelumnya (bila ada) sebagai konteks minimal.

Berikut contoh pembuatan request OpenAI dan mengirimkannya (dari fungsi `GetTenkiPlanFromLLM`):

```
var sys = BuildSystemPrompt();
var user = $"User said: {userText}.\nPrevious user's prompt: {previousUserText}
\n\nReturn ONLY JSON matching the schema.";
var reqObj = new ChatRequest {
    model = OpenAIModel,  // misal "gpt-4o-mini"
    messages = new[] {
        new ChatMessage{ role="system", content = sys },
        new ChatMessage{ role="user", content = user }
    },
    response_format = new ResponseFormat { type = "json_object" },
    temperature = 0.2f
};
var json = JsonUtility.ToJson(reqObj);
using (var req = new UnityWebRequest(OpenAIChatUrl, "POST"))
{
    byte[] body = Encoding.UTF8.GetBytes(json);
    req.uploadHandler = new UploadHandlerRaw(body);
    req.downloadHandler = new DownloadHandlerBuffer();
```

```
    req.SetRequestHeader("Content-Type", "application/json");
    req.SetRequestHeader("Authorization", "Bearer " + OpenAIApiKey);
    req.timeout = 30;

_inFlight.Add(req);              // catat request agar bisa dibatalkan jika
perlu
    yield return req.SendWebRequest();
    _inFlight.Remove(req);
    if (req.result != UnityWebRequest.Result.Success) {
        onError?.Invoke(req.error);
        yield break;
    }
    var text = req.downloadHandler.text;
    OpenAIResponse resp = JsonUtility.FromJson<OpenAIResponse>(text);
    // ... (ambil resp.choices[0].message.content lalu parse ke TenkiPlan)
}
```

Penjelasan:
- Kode di atas membentuk objek `ChatRequest` sesuai format API OpenAI, berisi pesan sistem dan pesan user. *response_format* diset `json_object` agar OpenAI API mencoba mengeluarkan JSON yang bisa langsung di-parse. Temperature dibuat rendah (0.2) agar model lebih deterministik sesuai instruksi. - Request dikirim sebagai HTTP POST ke `https://api.openai.com/v1/chat/completions` (atau ke URL *relay* jika `UseSecureRelay` = true). Kunci API disertakan di header Authorization. - Hasil dari `req.downloadHandler.text` diharapkan berupa JSON dari OpenAI, lalu di-mapping ke `OpenAIResponse` dan akhirnya mengambil `content` di pilihan pertama. - `content` ini seharusnya berupa string JSON sesuai skema TenkiPlan. Kode mencoba mengurai string JSON tersebut menjadi objek `TenkiPlan` melalui `JsonUtility.FromJson<TenkiPlan>(content)`. Jika ada kesalahan parsing (misal format sedikit melenceng), kode mencoba `Trim()` dan parse ulang, serta menangani error dengan mengirim onError callback. - Jika parsing sukses namun tidak ada `plan.intent`, maka itu dianggap gagal (model tidak mengikuti instruksi) – error ditangani sesuai itu.

Struktur TenkiPlan: (didefinisikan sebagai kelas dalam `MainProgram`)

```
[Serializable]
public class TenkiPlan {
    public string intent;            // "weather" atau "chitchat"
    public WeatherPlan weather_api;  // detail pemanggilan WeatherAPI (bisa
null)
    public PlanLocation location;    // info lokasi (bisa null atau incomplete)
    public PlanTime time;            // info waktu (bisa null)
    public string reply;             // balasan untuk chitchat (jika ada)
}
```

`WeatherPlan` memuat parameter seperti endpoint, query (`q`), jumlah hari, tanggal dsb. `PlanLocation` dan `PlanTime` memuat detail lokasi dan waktu yang dipahami model dari input user.

1.3. Menangani Intent Chitchat vs Weather

Setelah `TenkiPlan` berhasil diperoleh, *coroutine* `ProcessUserMessage` di `MainProgram` akan memeriksa nilai `plan.intent`:

- **Intent "chitchat":**
  Jika model menetapkan bahwa maksud pengguna bukan meminta cuaca, melainkan obrolan biasa, maka Tenki-Chan akan merespons dengan teks di properti `plan.reply`. Kode memastikan bahwa `plan.reply` terisi; jika kosong (seharusnya tidak kosong untuk chitchat), diisi dengan pesan default seperti *"Ayo ngobrol! ☺"*. Kemudian:
- Teks balasan tersebut ditampilkan melalui `SafeOutput(reply)`.
- Event `OnFinalReply?.Invoke(reply)` dipanggil (opsional, jika ada listener untuk final reply).
- Status UI diperbarui menjadi “Mengubah Teks ke Suara”.
- *Coroutine* TTS `TextToSpeechStart(reply, epoch)` dijalankan untuk mengonversi balasan teks menjadi suara yang diputar.

- Coroutine `ProcessUserMessage` **diakhiri** (`yield break`) karena alur untuk chitchat selesai di sini (tidak perlu panggil WeatherAPI).

- **Intent "weather":**
  Jika pengguna menanyakan cuaca, alurnya lebih panjang. Berikut tahapan utamanya:

- **Menentukan Query Lokasi (`q`):** Aplikasi memutuskan query string untuk WeatherAPI berdasarkan *plan* dan input user. Ini dilakukan fungsi `DeriveWeatherQ(plan, userText)`. Prioritasnya:
  - Jika `plan.weather_api.q` terisi (model mungkin sudah memberi saran query, misal `"Jakarta"` atau `"51.5072,-0.1276"`), gunakan itu.
  - Jika `plan.location` memiliki lat & lon yang valid (bukan 0), gunakan format `"lat,lon"`.
  - Jika `plan.location.query` ada, gunakan query tersebut.
  - Jika semua di atas tidak ada, gunakan teks input asli sebagai fallback.
- **Resolusi Nama Lokasi ke Koordinat:** Jika query `q` bukan dalam format `"lat,lon"`, maka kemungkinan `q` berupa nama tempat. Untuk meningkatkan akurasi, kode memanggil *WeatherAPI Search API* melalui coroutine `ResolveWithWeatherSearch(q, onResolved)`. Endpoint `search.json` WeatherAPI akan mengembalikan daftar lokasi matching; kode mengambil entri pertama dan membentuk koordinat `"lat,lon"` sebagai query baru `resolvedQ`. Jika search gagal atau kosong, tetap gunakan query awal.
  Catatan: Langkah ini memastikan misalnya input `"Kecamatan Dukun, Magelang"` dapat diubah menjadi koordinat spesifik sehingga data cuaca lebih tepat.
- **Menentukan Current vs Forecast:** Berdasarkan *plan* dan *time*:
  - Fungsi `ShouldUseForecast(plan)` akan mengecek apakah kita perlu panggil `forecast.json`. Kalau `plan.weather_api.endpoint` secara eksplisit adalah `"forecast"`, langsung *true*. Jika `plan.time.type` adalah `"date"` dan tanggal yang diminta di masa depan, atau `"relative"` (misal "besok", "lusa"), maka *true*. Untuk `"now"` atau tanggal hari ini, *false* (gunakan current).
  - Jika menggunakan forecast, tentukan berapa hari prakiraan diperlukan dengan `GetForecastDays(plan)` (default 3 hari jika relative time tidak jelas, atau 1 hari jika tanggal spesifik). Juga periksa jika ada tanggal tertentu `dt` (YYYY-MM-DD) dari *plan*.

- **Mengambil Data Cuaca dari WeatherAPI:**
  - Jika membutuhkan prakiraan (`useForecast == true` atau ada `dt`), *coroutine* `FetchForecast` dipanggil. Fungsi ini memanggil WeatherAPI `/forecast.json` dengan parameter *q*, *days*, *dt*. Untuk efisiensi, jika `UseSecureRelay` tidak aktif, kunci API disertakan langsung di URL; jika memakai relay, request dikirim ke URL relay (misal CloudFlare worker atau server user) yang bertugas meneruskan ke WeatherAPI dengan menambahkan kunci di sisi server.
  - Jika hanya cuaca saat ini, *coroutine* `FetchCurrent` dipanggil (endpoint `/current.json`). Keduanya bekerja mirip: mengirim HTTP GET, lalu mem-parse JSON ke dalam kelas *response* (`WeatherApiCurrentResponse` atau `WeatherApiForecastResponse`). Kelas-kelas ini didefinisikan untuk mencocokkan struktur JSON WeatherAPI.
  - Hasil parse dibungkus dalam objek `WeatherResult` yang menyimpan `location`, `current`, `forecast`, serta flag `isForecast`. Data penting seperti suhu, kondisi, kelembapan, dll tersedia dalam objek `current` atau di dalam `forecast.forecastday` untuk prakiraan.
  - Jika terjadi error (misal jaringan gagal), akan ditampilkan pesan error "Weather lookup failed".

Contoh kode memanggil WeatherAPI (cuplikan dari `FetchCurrent` dan `FetchForecast`):

```
string url = UseSecureRelay
    ? RelayBaseUrl.TrimEnd('/') + "/weatherapi/current?q=" +
UnityWebRequest.EscapeURL(q)
    : $"{WeatherApiBaseUrl}/current.json?key={WeatherApiKey}
&q={UnityWebRequest.EscapeURL(q)}&aqi=no";
using (var req = UnityWebRequest.Get(url))
{
    req.timeout = 30;
    _inFlight.Add(req);
    yield return req.SendWebRequest();
    _inFlight.Remove(req);
    if (req.result != UnityWebRequest.Result.Success) {
        onError?.Invoke(req.error);
        yield break;
    }
    var text = req.downloadHandler.text;
    WeatherApiCurrentResponse data =
JsonUtility.FromJson<WeatherApiCurrentResponse>(text);
    if (data == null || data.location == null || data.current == null) {
        onError?.Invoke("WeatherAPI current: missing fields");
        yield break;
    }
    onSuccess?.Invoke(new WeatherResult {
        location = data.location,
        current = data.current,
        forecast = null,
        isForecast = false
```

```
    });
}
```

Penjelasan:
- Contoh di atas untuk `FetchCurrent`: membangun URL request tergantung apakah melalui relay atau langsung. Parameter `q` sudah di-*escape*. `aqi=no` berarti data kualitas udara tidak diambil (tidak diperlukan). Timeout di-set 30 detik agar tidak hang terlalu lama.
- Request ditambahkan ke list `_inFlight` agar dapat dibatalkan jika user mengirim pertanyaan baru sebelum selesai. Setelah `yield return` (menunggu response), request dihapus dari tracking list.
- Hasil JSON di-*parse* ke `WeatherApiCurrentResponse` (kelas ini memiliki field `location` dan `current` sesuai respon WeatherAPI). Jika parsing gagal atau field penting kosong, anggap error.
- Jika sukses, panggil `onSuccess` callback dengan mengisi `WeatherResult`. Untuk *forecast*, proses serupa tapi data dimasukkan ke `WeatherResult.forecast` dan `isForecast = true`.

1. **Mengubah Data Cuaca Menjadi Kalimat (Natural Language):**
   Setelah mendapatkan `WeatherResult`, Tenki-Chan perlu menyampaikan informasi cuaca tersebut dengan gaya bahasa manusia yang ramah. Alih-alih menampilkan data mentah (suhu, kelembapan, dll), *MainProgram* memanfaatkan **OpenAI** sekali lagi untuk membuat narasi singkat. Fungsi coroutine `ResultToSpeakableText` mengirim *prompt* khusus berisi aturan format laporan cuaca. Prompt sistem dibuat di `BuildSystemPromptForWeather()`, yang isinya (garis besarnya):
2. Input: JSON *WeatherAPI response* akan diberikan (sebagai string) kepada model.
3. **Tugas model:** Keluarkan *script* ucapan (teks final) **tanpa format tambahan** (tanpa JSON, tanpa markup) yang akan dibacakan oleh Tenki-Chan. Hanya teks biasa dalam Bahasa Indonesia.
4. Gaya penyampaian: seperti pembawa acara ramalan cuaca, **jelas, ringkas, nada bersahabat tapi tidak berlebihan**. Panjang sekitar 80-160 kata.
5. **Konten:** Gunakan data *location* (nama lokasi, region, negara, zona waktu) untuk konteks tempat/waktu, gunakan unit metrik (Celcius, km/jam) karena di Indonesia umum dipakai, lakukan pembulatan angka (suhu & angin ke integer, curah hujan satu desimal jika <10). Tidak boleh mengada-adakan data yang tidak ada di respon WeatherAPI.
6. Semua output harus dalam **Bahasa Indonesia**.

Fungsi `ResultToSpeakableText` akan mengirim JSON `WeatherResult` yang sudah diperoleh sebagai bagian dari *user prompt*. Model OpenAI akan membalas dengan string (response_format di-set "text" agar tidak dikemas lagi dalam JSON). Keluaran ini adalah kalimat-kalimat siap ucap. Berikut cuplikan prosesnya:

```
var sys = BuildSystemPromptForWeather();
var user = $"User said:\n{JsonUtility.ToJson(res)}\n\nReturn ONLY speakable
script as plain text.";
var reqObj = new ChatRequest {
    model = OpenAIModel,
    messages = new[] {
        new ChatMessage{ role="system", content = sys },
        new ChatMessage{ role="user", content = user }
    },
    response_format = new ResponseFormat { type = "text" },
    temperature = 0.2f
```

```
};
var json = JsonUtility.ToJson(reqObj);
// ... (kirim request serupa dengan sebelumnya, parse OpenAIResponse)
string content = resp.choices[0].message.content;
onSuccess?.Invoke(content);
```

Setelah `finalText` (teks laporan cuaca) diperoleh, aplikasi: - Menampilkan data cuaca di UI secara terstruktur (melalui `OutputWeater()` – sedikit salah eja, seharusnya OutputWeather). Fungsi ini memperbarui tampilan dengan ikon cuaca, nama lokasi, suhu, kelembapan, dll. Contohnya:

```
weatherIconImage.sprite = WeatherIcons.GetSprite(result.current.condition.code,
result.current.is_day == 1);
locationText.text = plan?.location.query;
weatherConditionText.text = result.current.condition.text;
temperatureText.text = metric
    ? $"temp: {result.current.temp_c:0.#}°C"
    : $"temp: {result.current.temp_f:0.#}°F";
// ... dst untuk wind, humidity, uv, etc.
```

Kode di atas menampilkan ikon cuaca sesuai kode kondisi (pagi/malam), menampilkan nama lokasi yang ditanyakan, kondisi cuaca (cerah, mendung, dll), suhu dengan satu desimal, kecepatan angin, arah angin, kelembapan, UV, dll. *Note:* `metric` ditentukan berdasarkan unit yang diminta (default metric). - Mengisi `previousUserText` dengan teks final yang diucapkan (supaya jika pertanyaan berikutnya masih nyambung, bisa dijadikan konteks). - Mengosongkan input field (karena percakapan selesai untuk pertanyaan ini). - Melanjutkan ke tahap berikutnya (TTS).

1.4. Teks ke Suara (Text-to-Speech)

Tahap akhir untuk setiap respons Tenki-Chan adalah mengubah teks balasan menjadi suara, sehingga pengguna bisa mendengar Tenki-Chan berbicara. Ini ditangani oleh *coroutine* `TextToSpeechStart(string text, int epoch)`. Kode ini menggunakan API ElevenLabs (layanan *text-to-speech* berbasis AI) untuk menghasilkan audio dari input teks berbahasa Indonesia.

Berikut langkah-langkah yang dilakukan di `TextToSpeechStart`: 1. **Menyiapkan Payload JSON:** ElevenLabs API untuk *TTS* biasanya memerlukan teks dan parameter suara. Dalam kode, kelas `ElevenLabsVoice` didefinisikan dengan field `text` dan `language_code`. Suara yang digunakan ditentukan oleh *Voice ID* yang sudah disiapkan (misal `ElevenLabsVoiceId = "B8gJV1IhpuegLxdpXFOE"` dalam kode, yang mewakili karakter suara tertentu). Payload JSON dibuat dari objek ini:

```
var reqObj = new ElevenLabsVoice {
    text = text,
    language_code = "id"  // kode bahasa: id = Bahasa Indonesia
```

```
};
var json = JsonUtility.ToJson(reqObj);
```

2. **Mengirim Request TTS:** Membuat `UnityWebRequest` POST ke endpoint ElevenLabs (URL dasar `ElevenLabsBaseUrl` + `ElevenLabsVoiceId`). *UploadHandler* diisi dengan `json` di atas, *DownloadHandler* disiapkan untuk menerima data biner (audio). *Header* `"xi-api-key"` ditambahkan dengan kunci API ElevenLabs. Kemudian request dikirim:

```
UnityWebRequest request = new UnityWebRequest(ElevenLabsBaseUrl +
ElevenLabsVoiceId, "POST");
request.uploadHandler = new UploadHandlerRaw(bodyRaw);
request.downloadHandler = new DownloadHandlerBuffer();
request.SetRequestHeader("Content-Type", "application/json");
request.SetRequestHeader("xi-api-key", ElevenLabsApiKey);
request.timeout = 30;
yield return request.SendWebRequest();
```

3. **Menangani Respon Audio:** Jika request sukses, `request.downloadHandler.data` akan berisi byte audio (format MP3). Unity tidak bisa langsung mengubah byte MP3 menjadi audio di runtime tanpa plugin tambahan, jadi trik yang digunakan: - Menyimpan byte tersebut ke file sementara (misal `Application.persistentDataPath + "/tts.mp3"`). - Menggunakan `UnityWebRequestMultimedia.GetAudioClip("file://.../tts.mp3", AudioType.MPEG)` untuk memuat file MP3 tersebut sebagai `AudioClip`. - Setelah `yield return` selesai dan `www.result` sukses, mengambil `AudioClip clip = DownloadHandlerAudioClip.GetContent(www)`. - Memainkan clip tersebut dengan `audioSource.Play()`. (Sebelumnya memastikan `audioSource.clip = clip` dan jika ada clip lama di-stop). - Jika terjadi error pada langkah ini, akan dilog error, tapi eksekusi tetap berlanjut ke akhir. 4. **Mengaktifkan Kembali UI:** Setelah memulai pemutaran audio, fungsi `SetInteractable(true)` dipanggil untuk mengaktifkan kembali tombol kirim dan input field, sehingga pengguna bisa mengetik pertanyaan selanjutnya bahkan saat audio mungkin masih berjalan. `statusText` (teks status di UI) diubah menjadi `"Finishing"` menandakan satu siklus percakapan selesai diproses.

Cuplikan kode TTS dan pemutaran audio:

```
if (request.result == UnityWebRequest.Result.Success)
{
    byte[] audioData = request.downloadHandler.data;
    string tempPath = Application.persistentDataPath + "/tts.mp3";
    System.IO.File.WriteAllBytes(tempPath, audioData);
    using (UnityWebRequest www =
UnityWebRequestMultimedia.GetAudioClip("file://" + tempPath, AudioType.MPEG))
    {
        www.timeout = 30;
        yield return www.SendWebRequest();
        if (www.result == UnityWebRequest.Result.Success)
```

```
        {
            AudioClip clip = DownloadHandlerAudioClip.GetContent(www);
            audioSource.Stop();
            audioSource.clip = clip;
            audioSource.Play();
        }
        else
        {
            Debug.LogError("Failed to load audio: " + www.error);
        }
    }
}
else
{
    Debug.LogError("Error: " + request.error);
}
SetInteractable(true);
statusText.text = "Finishing";
```

Dapat dilihat bahwa setelah audio diputar, UI diaktifkan kembali tanpa menunggu audio selesai. Ini sengaja, supaya pengguna tidak perlu menunggu lama jika ingin langsung bertanya lagi. Audio akan tetap diputar hingga selesai di background.

1.5. Ringkasan Teknikal MainProgram

- **Coroutine & Async Flow:** Hampir semua operasi jaringan dijalankan sebagai Unity *coroutine* (`IEnumerator` dengan `yield return`). Ini menjaga aplikasi tetap responsif (penting di Unity terutama untuk WebGL, karena JavaScript tunggal thread). Setiap langkah – panggilan OpenAI, panggilan WeatherAPI, dsb – di-*yield* sehingga bisa berhenti sejenak sampai data datang, tanpa membekukan main thread.
- **Epoch & Request Cancellation:** Variabel `_epoch` dan `_inFlight` digunakan untuk **mengelola pembatalan operasi asinkron** jika pengguna mengirim pesan baru sebelum yang lama selesai. `_epoch` adalah versi dari "ID tugas" saat ini. Setiap memulai permintaan baru, epoch bertambah; setiap coroutine atau request memeriksa di akhirnya apakah epoch sudah berubah (berarti sudah kadaluarsa) dan akan `yield break` agar tidak melanjutkan output yang sudah tak relevan. List `_inFlight` menyimpan semua `UnityWebRequest` yang sedang berjalan; ketika memulai yang baru, semua request dalam list dibatalkan (`Abort`) untuk membebaskan koneksi dan memastikan tidak ada callback yang masuk dari permintaan lama.
- **Keamanan Kunci API:** Kode menunjukkan dua opsi: langsung memasukkan kunci API di *Inspector* Unity (`OpenAIApiKey`, `WeatherApiKey`, `ElevenLabsApiKey` di header "Keys") **atau** menggunakan *Secure Relay*. *Secure Relay* di sini maksudnya mengirim request ke server milik developer yang menambahkan kunci API di server side (jadi kunci tidak terpapar di klien). Misal, jika `UseSecureRelay = true` dan `RelayBaseUrl` terisi, maka `FetchCurrent` akan menggunakan URL relay seperti `https://yourserver.com/weatherapi/current?q=...` alih-alih langsung ke api.weatherapi.com dengan key. Demikian juga untuk OpenAI dan search.
- **Unity Services Cloud Code:** Pada `Start()` (fungsi Unity yang dipanggil saat awal), kode melakukan `await UnityServices.InitializeAsync()` dan `await`

`AuthenticationService.Instance.SignInAnonymouslyAsync()` lalu mencoba memanggil fungsi Cloud Code (melalui `MyModuleBindings`) untuk mengambil kunci API. Ini menunjukkan integrasi dengan Unity Cloud Code (mungkin ada script server yang menyimpan kunci-kunci dan mengembalikannya supaya tidak ditanam di klien). Jika berhasil, kunci API akan di-set ke variabel publik yang tadi disebut. (Jika gagal, akan tertangkap di `CloudCodeException` tapi tidak ada handling khusus selain log error).

- **Personality Tenki-Chan:** Variabel `TenkiPersona` (bisa diisi di Inspector, contoh: *"You are Tenki-Chan, a cheerful weather helper..."*) digunakan sebagai bagian dari system prompt. Hal ini menjadikan jawaban Tenki-Chan punya kepribadian ceria, ramah, dengan kata-kata sederhana. Meskipun begitu, format jawaban tetap dikontrol ketat agar output JSON sesuai skema.
- **UI Elements:** *MainProgram* mengelola peralihan tampilan antara mode obrolan biasa vs mode tampilan cuaca. Misal, ada dua panel `outputNormal` dan `outputWeather` yang diaktif/nonaktifkan tergantung apakah pertanyaan sedang menampilkan jawaban cuaca atau tidak (`outputNormal.SetActive(!isAskingWeather)`, `outputWeather.SetActive(isAskingWeather)`). Dengan demikian, ketika user bertanya cuaca, panel khusus dengan ikon dan detail cuaca muncul, sedangkan untuk obrolan biasa panel teks normal yang muncul.
- **Elemen Audio:** `audioSource` (komponen Unity `AudioSource`) digunakan untuk memutar suara hasil TTS. Volume, pitch, dsb., bisa diatur melalui komponen ini di editor Unity.

Setelah memahami komponen chatbot Tenki-Chan, selanjutnya mari kita lihat komponen kedua dalam repository ini, yaitu pengelola multithreading untuk pemrosesan cuaca massal.

2. MultithreadingManager – Pemrosesan Cuaca Massal Paralel

Skrip `MultithreadingManager.cs` bertanggung jawab untuk fitur mengunggah file (CSV/XLSX) berisi daftar lokasi dan mengambil data cuaca terkini untuk semua lokasi tersebut secara paralel. Ini sangat berguna misalnya untuk mendapatkan informasi cuaca terkini di banyak daerah sekaligus (contoh: daftar kecamatan dalam suatu kabupaten, sesuai nama kelas `KecamatanRow`). Meskipun namanya *MultithreadingManager*, implementasi paralelnya menggunakan *coroutine* Unity (karena Unity tidak mengizinkan threading bebas untuk sebagian besar Unity API), namun konsepnya serupa *multithreading* di mana banyak operasi I/O dilakukan serentak.

2.1. Alur Kerja Fitur Batch

1. **Memilih File Input:** Pengguna menekan tombol *Pick File* di UI. *MultithreadingManager* menangani klik ini dengan fungsi `OnPick()`, yang akan memicu mekanisme *file picker* di platform WebGL menggunakan fungsi *Javascript interop*. (Di WebGL, mengakses sistem file dilakukan via callback JS, di Editor Unity disediakan fallback).
2. **Memuat dan Mem-parse File:** Saat file dipilih oleh user, sebuah fungsi JS memanggil balik `OnWebFilePicked(string payload)` di script C#. *Payload* berisi nama file, ekstensi, dan data file (di-encode base64). `MultithreadingManager` kemudian:
3. Mendeteksi ekstensi (CSV atau XLSX) dari payload.
4. Jika CSV: mengonversi data ke string dan parsing CSV lewat fungsi `ParseCsv`.
5. Jika XLSX: mengonversi data base64 ke byte[] lalu parsing XLSX lewat `ParseXlsx`.
6. Hasil parsing mengisi list `_rows` dengan objek `KecamatanRow` (hanya dengan Name, Lat, Lon terisi awalnya).

7. Teks status UI diperbarui misal: `"Loaded 50 rows from data_kecamatan.xlsx"`.
8. **Memproses Data Cuaca:** Setelah file termuat, pengguna menekan tombol *Process*. Fungsi `OnProcess()` akan memvalidasi:
9. Apakah kunci WeatherAPI tersedia (harus diisi atau menggunakan relay, jika kosong akan menampilkan `"WeatherAPI key required."`).
10. Apakah ada data lokasi di `_rows`. Jika belum, ia mencoba fallback: jika dijalankan di Editor/ Standalone, akan mencari file `StreamingAssets/input.csv` dan memuatnya (ini untuk memudahkan pengujian di editor tanpa UI upload). Jika tetap tidak ada data, tampilkan pesan `"No rows to process. Upload a file first."`.
11. Jika semua ok, dia memanggil `StopAllCoroutines()` (untuk jaga-jaga tidak ada proses jalan) lalu memulai *coroutine* utama `ProcessAll()`.
12. **Pengambilan Cuaca Paralel:** Fungsi `ProcessAll()` inilah inti *multithreading* dengan *coroutine*. Ia membuat sejumlah *worker coroutines* sesuai `maxConcurrency` (jumlah maksimum request bersamaan, default di code 48). Masing-masing worker mengambil satu lokasi dari list `_rows`, melakukan request WeatherAPI (sama seperti `FetchCurrent` pada MainProgram, tapi di sini langsung diimplementasi di fungsi `FillWeather`), lalu mengambil lokasi berikutnya, sampai semua lokasi selesai diproses. Semua ini dilakukan tanpa menunggu satu-per-satu secara berurutan, melainkan banyak request berjalan bersama untuk mempercepat total waktu.
13. **Unduh Hasil:** Setelah semua selesai (`ProcessAll` selesai), pengguna dapat menekan tombol *Download CSV* atau *Download XLSX*. Fungsi `Download(string kind)` akan mengambil data hasil (_rows yang kini sudah terisi informasi cuaca) dan membangun string CSV atau file XLSX, lalu memanfaatkan fungsi *bridge* ke JavaScript untuk menyimpan file tersebut ke komputer pengguna (atau men-trigger download di browser, dalam kasus WebGL).

Berikut diagram sederhana (tanpa gambar) bagaimana *MultithreadingManager* berjalan: - **Upload file → Parse → [Data lokasi1, lokasi2, ...]**
- **Process (parallel)** → untuk setiap lokasi lakukan GET cuaca (banyak sekaligus) → isi data cuaca di setiap entri.
- **Download** → simpan ke file hasil.

Selanjutnya, kita akan membahas implementasi teknis penting dari komponen ini.

2.2. Memuat dan Parsing File (CSV/XLSX)

Pada saat file dipilih, fungsi `OnWebFilePicked(string payload)` akan menangani data file. *Payload* berupa string dengan format `"namaFile.ext|ext|<dataBase64>"`. Kode memecah string ini dengan mencari tanda `|` pertama dan kedua, untuk memisahkan nama file, ekstensi, dan data. Cuplikan:

```
public void OnWebFilePicked(string payload)
{
    try {
        // Pisahkan komponen payload
        var firstPipe = payload.IndexOf('|');
        var secondPipe = payload.IndexOf('|', firstPipe + 1);
        var filename = payload.Substring(0, firstPipe);
        var ext = payload.Substring(firstPipe + 1, secondPipe - firstPipe -
```

```
1).ToLowerInvariant();
        var b64 = payload.Substring(secondPipe + 1);
        // Simpan nama file (tanpa ekstensi) dan jenis
        _sourceFilename = Path.GetFileNameWithoutExtension(filename);
        _sourceKind = (ext == "xlsx") ? "xlsx" : "csv";
        // Konversi base64 ke bytes
        var bytes = Convert.FromBase64String(b64);
        if (_sourceKind == "csv") {
            var csv = Encoding.UTF8.GetString(bytes);
            ParseCsv(csv);
        } else {
            ParseXlsx(bytes);
        }
        statusText.text = $"Loaded {_rows.Count} rows from {filename}.";
    } catch (Exception ex) {
        statusText.text = $"Failed to load file: {ex.Message}";
        Debug.LogException(ex);
    }
}
```

Penjelasan:
- Setelah mendapatkan `bytes` dari base64, jika file adalah CSV, diubah menjadi string dengan asumsi encoding UTF8 lalu diproses `ParseCsv(csv)`. Jika XLSX, langsung kirim byte array ke `ParseXlsx`. - `_sourceFilename` dan `_sourceKind` disimpan untuk keperluan penamaan file hasil nantinya. - Variabel `_rows` (List of `KecamatanRow`) diisi di fungsi parse. `KecamatanRow` adalah kelas internal yang merepresentasikan satu baris lokasi, dengan property: `Name, Lat, Lon` (input) dan kolom-kolom hasil: `LastUpdate, SuhuC, Kelembapan, Kondisi, KecepatanAnginKph, ArahAngin, UV`. - Jika parsing gagal (misal format file salah), akan muncul pesan error dan di `Debug.LogException` untuk developer.

Parsing CSV:
Fungsi `ParseCsv(string csv)` membaca baris pertama sebagai header, mencari indeks kolom yang sesuai. Kode menggunakan regex pada string `nameHeader`, `latHeader`, `lonHeader` untuk mencocokkan nama-nama kolom (case-insensitive, mendukung beberapa kemungkinan nama misal "nama|name|kecamatan" untuk kolom nama, "lat|latitude|lintang" untuk lintang, dll). Jika tidak menemukan kolom yang diperlukan, lempar Exception agar pengguna tahu format tidak cocok. Kemudian setiap baris diolah:

```
using var reader = new StringReader(csv);
string header = reader.ReadLine();
var headers = SplitCsvLine(header).ToArray();
int idxName = FindIndex(headers, nameHeader);
int idxLat  = FindIndex(headers, latHeader);
int idxLon  = FindIndex(headers, lonHeader);
if (idxName < 0 || idxLat < 0 || idxLon < 0)
    throw new Exception("Input CSV must contain columns: Name, Lat, Lon ...");
for (var line = reader.ReadLine(); line != null; line = reader.ReadLine()) {
```

```
    if (string.IsNullOrWhiteSpace(line)) continue;
    var cols = SplitCsvLine(line).ToArray();
    if (cols.Length <= Math.Max(idxName, Math.Max(idxLat, idxLon))) continue;
    var row = new KecamatanRow {
        Name = cols[idxName].Trim(),
        Lat  = ParseDouble(cols[idxLat]),
        Lon  = ParseDouble(cols[idxLon])
    };
    _rows.Add(row);
}
```

Penjelasan:
- `SplitCsvLine` adalah fungsi pembantu untuk membagi satu baris CSV menjadi kolom-kolom dengan memperhatikan tanda kutip ganda. (Isi di dalam `"` yang mungkin mengandung koma harus tetap utuh sebagai satu kolom). - `FindIndex(headers, pattern)` mencoba mencari indeks kolom yang namanya sesuai *pattern* regex. Pertama kecocokan penuh (`^(?:pattern)$`), jika tidak, mencari substring mengandung pattern (agar sedikit fleksibel). - Setiap baris data diubah ke `KecamatanRow`. Fungsi `ParseDouble` digunakan untuk parse string ke double dengan CultureInvariant (mengganti koma dengan titik jika perlu). - Akhirnya, `_rows` akan berisi list lokasi dengan koordinat siap.

Parsing XLSX:
Untuk file Excel (.xlsx), kode melakukan parsing manual *tanpa library Excel eksternal*. Ini dilakukan dengan sedikit trik: - File XLSX adalah file *zip* berisi file XML. Kode membuka byte array XLSX dengan `ZipArchive`. - Membaca file `xl/sharedStrings.xml` jika ada (berisi semua string teks unik di dokumen, digunakan karena Excel menyimpan teks secara terpisah untuk menghemat ukuran). - Membaca file `xl/worksheets/sheet1.xml` (atau sheet pertama yang ditemukan). Lalu dengan regex, mencari setiap elemen baris `<row>` dan tiap sel `<c ...>` di dalamnya. - Setiap sel `<c>` punya attribute `r` (contoh `r="A1"`, `A` menandakan kolom, `1` baris), attribute `t` bisa `"s"` (string) artinya nilai ada di sharedStrings, atau default (angka). Elemen dalam `<c>` berisi `<v>...</v>` untuk nilai. - Kode menerjemahkan kolom huruf (A, B, AA, dll) menjadi indeks angka kolom (A->0, B->1, ..., Z->25, AA->26, dst). - Mengisi struktur `rows` (List of List<string>) di memori, di mana `rows[0]` berisi header. - Setelah terbentuk mirip dengan hasil CSV (list of row values), proses identifikasi kolom Name, Lat, Lon sama seperti tadi, mengisi `_rows`.

Parsing XLSX cukup kompleks, namun intinya mencari text di `sharedStrings` jika sebuah cell merujuk ke index string, atau mengambil angka langsung. Jika ada cell kosong yang terlewat di XML (Excel tidak menyimpan cell kosong dalam XML), kode mengisi string kosong di posisi tersebut agar indexing kolom tetap konsisten.

Kesimpulan parsing: Setelah tahap ini, `_rows` (List<KecamatanRow>) siap dengan ratusan atau ribuan lokasi yang akan diproses. Selanjutnya adalah bagian terpenting: pemrosesan paralel untuk mengisi data cuaca.

2.3. Pemrosesan Paralel Data Cuaca dengan Coroutine

Bagian ini adalah inti dari *MultithreadingManager*. Tujuannya adalah melakukan banyak request ke WeatherAPI (cuaca saat ini untuk setiap koordinat) secara bersamaan agar cepat selesai. Kode memanfaatkan kemampuan Unity menjalankan banyak coroutine parallel.

Lihat fungsi `ProcessAll()` berikut:

```
IEnumerator ProcessAll()
{
    statusText.text = $"Processing {_rows.Count} rows…";
    var startTime = Time.realtimeSinceStartup;
    int completed = 0;
    int idx = -1;
    int N = Mathf.Max(1, maxConcurrency);
    var workers = new List<Coroutine>(N);
    // Luncurkan N "worker" coroutine
    for (int i = 0; i < N; i++)
        workers.Add(StartCoroutine(Worker()));
    // Tunggu semua worker selesai
    foreach (var w in workers) yield return w;
    var dur = Time.realtimeSinceStartup - startTime;
    statusText.text = $"Done: {completed}/{_rows.Count} rows in {dur:0.0}s";
    // Inner coroutine Worker
    IEnumerator Worker()
    {
        while (true) {
            int my = System.Threading.Interlocked.Increment(ref idx);
            if (my >= _rows.Count) yield break;
            var row = _rows[my];
            yield return StartCoroutine(FillWeather(row));
            int c = System.Threading.Interlocked.Increment(ref completed);
            if (c % 50 == 0 || c == _rows.Count)
                statusText.text = $"Processed {c}/{_rows.Count}";
        }
    }
}
```

Penjelasan:
- `maxConcurrency` diatur default 48 (bisa diubah di Inspector). Misal ada 1000 lokasi dan maxConcurrency = 48, artinya 48 permintaan cuaca akan dikirim bersamaan. - `idx` adalah indeks global lokasi yang sudah atau sedang diproses. `completed` jumlah yang sudah selesai diisi datanya. - Diluncurkan `N` buah coroutine *Worker()* dengan `StartCoroutine`. Masing-masing *Worker* mengambil indeks berikutnya menggunakan `Interlocked.Increment(ref idx)`. Penggunaan `Interlocked.Increment` memastikan operasi penambahan atomik (karena walau single-thread, beberapa coroutine bisa hampir bersamaan mengambil index). Nilai `my` yang didapat adalah indeks ke

data berikutnya untuk diproses. - Jika `my >= _rows.Count`, artinya tidak ada lagi data, maka *worker* tersebut `yield break` (keluar dari loop dan coroutine berakhir). - Jika ada, ia mengambil `row = _rows[my]` dan memulai coroutine `FillWeather(row)`. `yield return StartCoroutine` digunakan agar *worker* menunggu sampai `FillWeather` selesai untuk satu lokasi, baru melanjutkan loop mengambil lokasi berikutnya. - Setelah `FillWeather` selesai, variabel `completed` di-increment atomik, dan setiap kelipatan 50 atau di akhir, `statusText` UI diperbarui (misal "Processed 150/1000"). - Loop *while(true)* pada *Worker* memastikan setiap *worker* akan terus mengambil tugas baru sampai habis. - Sementara para *worker* berjalan, `ProcessAll` sendiri melakukan `yield return w` pada tiap worker coroutine, sehingga `ProcessAll` akan selesai *setelah semua* worker selesai (sinkronisasi akhir). Barulah durasi dihitung dan status "Done: X/Y rows in Z sec" ditampilkan. - Dengan teknik di atas, jika misal ada 1000 lokasi dan 50 concurrency, total waktu mendekati `1000/50 = 20` kali durasi satu request (dengan asumsi load seimbang). Tanpa concurrency, akan 1000 kali lebih lambat.

Mengisi Cuaca per Baris (FillWeather):
Coroutine `FillWeather(KecamatanRow row)` melakukan tugas mirip `FetchCurrent` di MainProgram, tetapi juga dengan *retry logic*. Karena dalam batch mungkin terjadi kegagalan jaringan atau rate limit, kode mengatur hingga `maxRetries` percobaan per lokasi.

```
IEnumerator FillWeather(KecamatanRow row)
{
    string q = $"{row.Lat.ToString(CultureInfo.InvariantCulture)},
{row.Lon.ToString(CultureInfo.InvariantCulture)}";
    int attempt = 0;
    while (true) {
        attempt++;
        UnityWebRequest req;
        if (useRelay) {
            var url = $"{relayBaseUrl.TrimEnd('/')}/weatherapi/current?
q={UnityWebRequest.EscapeURL(q)}";
            req = UnityWebRequest.Get(url);
        } else {
            var key = tenkiChatController.WeatherApiKey.Trim();
            var url = $"{weatherApiBaseUrl}/current.json?key={key}
&q={UnityWebRequest.EscapeURL(q)}&aqi=no";
            req = UnityWebRequest.Get(url);
        }
        req.timeout = 30;
        yield return req.SendWebRequest();
        bool transient = req.result != UnityWebRequest.Result.Success
                        || req.responseCode == 429
                        || (req.responseCode >= 500 && req.responseCode < 600);
        if (!transient) {
            try {
                var json = req.downloadHandler.text;
                var data =
JsonUtility.FromJson<WeatherApiCurrentResponse>(json);
```

```
                if (data?.current == null) throw new Exception("Missing
current.");
                // Isi data ke row
                row.LastUpdate       = data.current.last_updated ?? "";
                row.SuhuC            = data.current.temp_c;
                row.Kelembapan       = data.current.humidity;
                row.Kondisi          = data.current.condition?.text ?? "";
                row.KecepatanAnginKph= data.current.wind_kph;
                row.ArahAngin        =
string.IsNullOrEmpty(data.current.wind_dir)
                                        ? $"{data.current.wind_degree}°"
                                        : data.current.wind_dir;
                row.UV               = data.current.uv;
            } catch (Exception ex) {
                if (verbose) Debug.LogWarning($"Parse error for {row.Name}:
{ex.Message}");
            }
            yield break; // berhasil (atau parse error tapi bukan transient),
keluar loop
        }
        // Jika error transient (misal 429 rate limit atau jaringan), cek retry
        if (attempt >= maxRetries) {
            if (verbose)
Debug.LogWarning($"Giving up {row.Name} after {attempt} tries. HTTP
{req.responseCode} {req.error}");
            yield break;
        }
        // Tunggu beberapa waktu sebelum retry (exponential backoff dengan
jitter)
        float delay = Mathf.Min(10f, 0.8f * attempt +
UnityEngine.Random.Range(0f, 0.6f));
        yield return new WaitForSecondsRealtime(delay);
        // loop lagi
    }
}
```

Penjelasan:
- Variabel `q` langsung dibuat dari Lat,Lon (karena kita sudah punya koordinat dari file, tidak perlu nama lokasi). - Membentuk UnityWebRequest `req`. Jika `useRelay == true`, maka akan panggil via URL relay (server developer) seperti `https://myrelay.example.com/weatherapi/current?...`. Jika `false`, langsung ke `weatherApiBaseUrl` (default `"https://api.weatherapi.com/v1"`). Kunci API diambil dari `tenkiChatController.WeatherApiKey` – di sini `MultithreadingManager` memiliki referensi ke `MainProgram` sebagai `tenkiChatController`, sehingga bisa memanfaatkan kunci API yang sama. (Ini lebih sederhana ketimbang duplikat variabel kunci). - `yield return req.SendWebRequest();` menunggu sampai request selesai (sukses/gagal). - Variabel `transient` menentukan apakah error-nya sebaiknya di-*retry*. Kondisi *transient* di sini meliputi: - `req.result != Success` (misal koneksi gagal, timeout). - HTTP response code 429 (Too Many Requests / dibatasi rate limit). - HTTP 5xx (server error di

WeatherAPI). Jika *transient* == *false*, artinya permintaan berhasil (200 OK) **atau** gagal tapi bukan jenis yang bisa ditangani dengan coba ulang (misal 401 Unauthorized – key salah, 400 Bad Request – koordinat tak valid, dll). Dalam kasus non-transient, tidak ada gunanya retry. - Jika bukan transient: - Coba parse JSON ke `WeatherApiCurrentResponse`. Jika `data.current` ada, isi properti-properti di `row` (LastUpdate, SuhuC, dll) dari `data.current`. Juga kondisi cuaca (teks) dan arah angin: jika `wind_dir` kosong, gunakan angka derajat `wind_degree` dengan simbol °, jika ada, pakai singkatan `wind_dir` (misal "NW"). - Jika parsing JSON gagal (exception tertangkap), akan log warning (jika `verbose` true). Meskipun gagal parse, kita anggap permintaan ini selesai (mungkin format berubah, tetapi jangan ulangi karena status code sukses). - `yield break` untuk mengakhiri coroutine `FillWeather` (keluar loop). - Jika *transient* == *true* (contohnya, dapat HTTP 429 rate limit): - Cek apakah `attempt` sudah mencapai `maxRetries`. Jika ya, log "Giving up {name} after X tries" (jika verbose) dan `yield break` (berhenti, tidak bisa dapat data). - Jika masih ada jatah retry, hitung `delay` untuk tunggu sebelum mencoba lagi. Delay dihitung dengan komponen linear `0.8 * attempt` detik + sedikit random `0 to 0.6` detik, dibatasi maksimum 10 detik. Pola ini memberikan *exponential backoff* ringan agar tidak spam server jika terus gagal. - `yield return new WaitForSecondsRealtime(delay);` menunggu selama itu (Realtime karena kita ingin jeda bahkan jika timescale game 0 atau di background). - Lalu loop `while(true)` berlanjut ke attempt berikutnya.

Dengan algoritma di atas, setiap lokasi akan dicoba hingga misalnya 3 kali sebelum menyerah. Semua ini berjalan paralel untuk banyak lokasi berkat mekanisme di `ProcessAll` yang menjalankan banyak coroutine *FillWeather* via *Worker* concurrently.

2.4. Mengunduh dan Menyimpan Hasil (CSV & XLSX)

Setelah `ProcessAll` selesai, list `_rows` kini sudah terisi semua data cuaca. Pengguna dapat mengunduh hasilnya. Dua tombol disediakan: CSV dan XLSX. Keduanya ditangani oleh fungsi `Download(string kind)`.

Download CSV:
Jika `kind == "csv"`, kode memanggil `BuildCsv()` untuk membentuk string CSV. `BuildCsv` membuat header kolom (Name, Latitude, Longitude, Last Update, Suhu (°C), Kelembapan (%), Kondisi, Kecepatan Angin (kph), Arah Angin, Sinar UV). Lalu setiap baris di `_rows` ditambahkan:

```
sb.Append(Escape(r.Name)).Append(',')
  .Append(r.Lat.ToString(CultureInfo.InvariantCulture)).Append(',')
  .Append(r.Lon.ToString(CultureInfo.InvariantCulture)).Append(',')
  .Append(Escape(r.LastUpdate)).Append(',')
  .Append(r.SuhuC.ToString("0.#", CultureInfo.InvariantCulture)).Append(',')
  .Append(r.Kelembapan).Append(',')
  .Append(Escape(r.Kondisi)).Append(',')
  .Append(r.KecepatanAnginKph.ToString("0.#",
CultureInfo.InvariantCulture)).Append(',')
  .Append(Escape(r.ArahAngin)).Append(',')
  .Append(r.UV.ToString("0.#", CultureInfo.InvariantCulture)).AppendLine();
```

Fungsi `Escape(string v)` menambahkan tanda kutip ganda jika diperlukan (kalau teks mengandung koma atau kutip atau newline). Angka diformat menggunakan titik desimal (`CultureInfo.InvariantCulture`) dan satu desimal untuk Suhu, Kecepatan Angin, UV (0.# artinya tanpa desimal kalau bulat, atau satu angka desimal jika ada pecahan).

Setelah string CSV terbentuk, `Download` akan memanggil `FileBridge_DownloadText(...)` dengan MIME "text/csv". Pada platform WebGL, `FileBridge_DownloadText` sudah di-`DllImport("__Internal")` yang berarti itu adalah fungsi JavaScript yang akan membuat file dan trigger download. Di Editor, fallback-nya menulis file ke disk proyek (untuk kemudahan debug).

Download XLSX:
Proses pembuatan XLSX lebih rumit karena melibatkan membentuk file Excel dari nol. Berikut ringkasannya: - `BuildXlsx()` mengumpulkan data ke struktur `rows` mirip saat parsing, tapi sekarang termasuk header dan semua data. - Ia membuat list `sst` (shared strings) untuk menampung teks unik. - Lalu membentuk string XML untuk worksheet: membuka `<worksheet><sheetData>`, lalu untuk setiap baris (dengan index r mulai 1): - Buka `<row r="i">`. - Untuk setiap kolom c dalam baris: hitung alamat A1-style (misal A1, B1, ..., AA1, dll) via fungsi `ToA1(colIndex)`. - Jika baris pertama (header) atau kolom tertentu yang diinginkan dianggap teks, simpan nilai di sharedStrings (`<c r="A1" t="s"><v>index_di_sharedStrings</v></c>`). - Untuk kolom angka, masukkan langsung `<c r="B2"><v>value</v></c>` tanpa `t="s"`. - Tutup `</row>`. - Tutup `</sheetData></worksheet>`. - Buat XML untuk sharedStrings `<sst count=.. uniqueCount=..> ... </sst>` dengan setiap string diapit `<si><t>...</t></si>`. - Buat juga file XML standar XLSX lain: `[Content_Types].xml`, `xl/workbook.xml`, relationships, `docProps` minimal. Kode sudah menyiapkan string untuk masing-masing. - Gunakan `ZipArchive` (mode Create) untuk menulis setiap file XML tersebut ke entri zip dengan path yang tepat (misal `xl/worksheets/sheet1.xml`, `xl/sharedStrings.xml`, dsb). - Keluarkan `MemoryStream` zip sebagai byte[].

Setelah mendapat `xlsxBytes`, fungsi `Download` melakukan:

```
var b64 = Convert.ToBase64String(xlsxBytes);
var binary = Base64ToLatin1(b64);
FileBridge_DownloadText($"{_sourceFilename}_weather.xlsx",
    "application/vnd.openxmlformats-officedocument.spreadsheetml.sheet",
    binary);
statusText.text = "XLSX downloaded.";
```

Kenapa di-convert ke Latin1 string? Karena `FileBridge_DownloadText` diasumsikan menerima string. Agar byte biner dapat terkirim utuh, mereka encode base64 lalu decode ke string 8-bit (Latin1) sehingga setiap karakter 0-255 sesuai byte asli. Lalu di sisi JS, tampaknya fungsi JS akan membuat Blob dengan MIME given dan data string tersebut. (Detail implementasi JS tidak ditampilkan di kode C#, tapi komentar menyebut cara ini berhasil membuat file terunduh dengan benar).

2.5. Integrasi UI dan Komponen Lain

Beberapa hal tambahan mengenai *MultithreadingManager*: - Di `Awake()`, skrip mengaitkan tombol UI ke fungsinya:

```
if (btnPick) btnPick.onClick.AddListener(OnPick);
if (btnProcess) btnProcess.onClick.AddListener(OnProcess);
if (btnDownloadCsv) btnDownloadCsv.onClick.AddListener(() => Download("csv"));
if (btnDownloadXlsx) btnDownloadXlsx.onClick.AddListener(() =>
Download("xlsx"));
statusText.text = "Ready. Upload a .csv or .xlsx.";
```

`ButtonManager` di sini kemungkinan adalah kelas UI (mungkin dari Michsky.DreamOS UI kit) yang mirip Button. Intinya, klik tombol akan memicu proses yang sudah dijelaskan. - Terdapat referensi `public MainProgram tenkiChatController;` yang dihubungkan di Inspector. Ini untuk mengakses `WeatherApiKey` (dan mungkin memanfaatkan instance yang sama untuk hal lain jika perlu). Dalam implementasi, `OnProcess` menggunakan `tenkiChatController.WeatherApiKey` supaya tak perlu memasukkan kunci dua kali. - `useRelay`, `relayBaseUrl`: Pengaturan ini serupa konsep di MainProgram untuk opsi mengarahkan request WeatherAPI lewat server sendiri. - `maxConcurrency` dan `maxRetries` dapat diatur di Inspector untuk mengubah perilaku paralel. Pastikan tidak terlalu besar melebihi limit environment (misal WebGL mungkin punya batas ~100 concurrent connections). - `verbose`: jika diaktifkan, log tambahan akan dicetak, seperti peringatan saat parse JSON gagal per lokasi, atau saat sebuah lokasi sudah dicoba maksimum tanpa hasil.

2.6. Contoh Penggunaan Fitur Batch

Misalkan kita memiliki file `daftar_kecamatan.csv` dengan isi:

```
Nama,Lat,Lon
Kecamatan Dukun, -7.329, 110.318
Kecamatan Mungkid, -7.585, 110.269
Kecamatan Borobudur, -7.607, 110.203
...
```

Langkah penggunaan: - Jalankan aplikasi (misal di WebGL di browser). - Klik **"Pick File"** dan pilih `daftar_kecamatan.csv`. - Status akan menampilkan berapa baris dimuat, misal "Loaded 3 rows from daftar_kecamatan.csv." - Klik **"Process"**. Sistem akan mulai mengambil data cuaca untuk 3 lokasi tersebut secara paralel. Jika `maxConcurrency` >= 3, maka ketiga request berjalan bersamaan. - Selama proses, status menampilkan "Processed X/3" seiring lokasi selesai. - Setelah selesai, status menjadi "Done: 3/3 rows in 1.2s" (misalnya). - Kemudian bisa klik **"Download CSV"** untuk mengunduh file `daftar_kecamatan_weather.csv` yang berisi kolom tambahan (Last Update, Suhu, dll) dengan data terbaru. Atau klik **"Download XLSX"** untuk file Excel serupa. - File hasil dapat dibuka untuk melihat informasi cuaca masing-masing kecamatan.

3. Kesimpulan

Tenki-Source menggabungkan teknologi AI dan data cuaca dalam aplikasi Unity yang interaktif. Komponen **MainProgram** menyediakan chatbot Tenki-Chan yang mampu memahami bahasa alami (dengan bantuan OpenAI) dan memberi jawaban cuaca aktual (dengan WeatherAPI) serta mengekspresikannya secara lisan (dengan ElevenLabs TTS). Di sisi lain, komponen **MultithreadingManager** menunjukkan bagaimana Unity bisa menangani *multithreading* (dengan coroutine) untuk melakukan ratusan permintaan jaringan secara paralel, sehingga pengambilan data cuaca skala besar dapat dilakukan dengan cepat.

Beberapa poin teknis penting yang bisa dipelajari dari repository ini: - **Integrasi API eksternal di Unity:** Menggunakan `UnityWebRequest` untuk REST API calls (dengan JSON), termasuk metode pengelolaan header, body, dan penanganan async dengan coroutine. - **Penggunaan Coroutine untuk Pipeline Asinkron:** Mengatur rangkaian langkah (LLM → WeatherAPI → LLM → TTS) dalam satu fungsi coroutine mempermudah alur pemrograman asynchronous tanpa callback hell. - **Pembatalan Operasi Asinkron:** Teknik menggunakan token (epoch) dan membatalkan request lama memastikan respons yang datang selalu sesuai dengan pertanyaan terakhir user, meningkatkan pengalaman pengguna. - **Pemrosesan Paralel (Multi-Coroutine):** Memanfaatkan banyak coroutine serentak untuk mempercepat tugas I/O yang berulang. Pola yang ditunjukkan dengan `Interlocked.Increment` dan loop *Worker* dapat diterapkan untuk skenario serupa (misal memuat banyak URL gambar, dsb.). - **Manipulasi File CSV/XLSX:** Contoh membaca file CSV dan Excel di Unity, termasuk trik membuat file Excel sendiri. Hal ini berguna jika ingin ekspor data dalam format spreadsheet tanpa bergantung library besar.

Semoga penjelasan di atas membantu memahami kode dalam repository **Tenki-Source** ini. Dengan README ini, pengembang maupun pengguna dapat lebih mudah mengerti tujuan dan cara kerja komponen Tenki-Chan dan fitur batch cuaca, baik dari level gambaran umum hingga detail teknis implementasinya. Selamat bereksperimen dengan Tenki-Chan!
