Project: Gmail “Insert Availability” Chrome Extension

0) What we are building

A Chrome extension that runs on `mail.google.com`. When a user is composing an email in Gmail, the extension injects a button (e.g., **Insert Availability**) into the Gmail compose toolbar. Clicking it opens a small UI (modal) where the user enters:

- Date range: `1 week / 2 weeks / custom`
- Daily working window: start time + end time (in the user’s “source” timezone)
- Minimum meeting duration (e.g., 30/45/60 minutes)
- Days of week (weekdays default)
- **User/source timezone** (the timezone their calendar + preferences are in)
- **Output timezone** (timezone that should be used in the inserted email text)

Then the extension:

1. Authenticates with Google via OAuth (no private keys).
2. Calls Google Calendar **FreeBusy** API for the chosen range.
3. Computes free time blocks within the daily window by subtracting busy intervals.
4. Filters blocks shorter than `min duration`.
5. Converts final blocks to **output timezone** for formatting.
6. Inserts a formatted text block into the current Gmail draft body at the cursor.

Non-goals / constraints for v1

- No backend server.
- No secrets.
- No persistent storage required (no localStorage / chrome.storage).
  User enters settings each time they click Insert Availability.
- Use FreeBusy only (don’t need event titles).
- Support primary calendar only (v1).

---

1) User experience flow (UX)

Gmail compose experience

- User opens a Gmail compose window.
- Extension injects a button into compose toolbar: **Insert Availability**.
- On click: open an overlay modal inside Gmail (simple HTML/CSS injected).
- Modal fields:
  - Range: “Next 1 week”, “Next 2 weeks”, “Custom start/end date”
  - Daily window: Start time, End time
  - Min duration: dropdown (30/45/60/90)
  - Days: checkboxes Mon–Sun (default Mon–Fri)
  - Source TZ: text input or dropdown (IANA timezone like `America/Chicago`)
  - Output TZ: text input or dropdown (IANA timezone like `America/New_York`)
  - “Generate & Insert”
- If not authenticated:
  - Trigger OAuth automatically after user clicks “Generate & Insert”
  - Then proceed

Inserted email text format (example)

Use output timezone for display:

Here are a few times that work for me over the next 2 weeks (ET):

Mon, Jan 12: 10:00 AM–12:30 PM, 2:00 PM–5:00 PM
Tue, Jan 13: 9:00 AM–11:00 AM
Thu, Jan 15: 1:00 PM–4:00 PM

If none of these work, feel free to share a few times that do.

---

2) Architecture overview (MV3)

Components

1. **Content Script** (runs in Gmail page)
- Detect compose windows
- Inject “Insert Availability” button
- Render the modal UI
- Collect user inputs
- Send a message to background service worker to fetch availability
- Receive computed formatted text
- Insert it into Gmail draft body
2. **Background Service Worker** (MV3)
- Handles OAuth token acquisition via `chrome.identity`

- Calls Google Calendar FreeBusy API
- Runs slot-generation algorithm
- Returns final formatted string to content script

Why split like this

- Content scripts are best for DOM + Gmail injection.
- Background is best for OAuth + network calls + heavier logic.

---

3) File structure

```
gmail-availability-extension/
  manifest.json
  src/
    background/
      service_worker.js
      google_auth.js
      calendar_api.js
      availability.js
      time_utils.js
      format.js
    content/
      content_script.js
      gmail_dom.js
      modal_ui.js
      modal_ui.css
      insert_text.js
    shared/
      constants.js
  assets/
    icon16.png
    icon48.png
    icon128.png
```

---

4) manifest.json (MV3 requirements)

Needs:

- `manifest_version: 3`

- permissions:
  - `identity` (OAuth token)
  - `activeTab` (safe default)
  - `scripting` (if needed; but content_scripts usually enough)
- host_permissions:
  - `https://mail.google.com/*`
  - `https://www.googleapis.com/*`
- content script matches Gmail
- background service worker

Important: OAuth in extensions uses a Google Cloud OAuth client with a Chrome extension redirect. Configure properly in Google Cloud Console.

---

5) Google Cloud setup (OAuth) — no secrets

In Google Cloud Console:

- Create a project
- Enable **Google Calendar API**
- Create OAuth client for **Chrome Extension** (or "Web app" depending on identity flow approach, but prefer Chrome extension identity)
- Get **Client ID**
- Add authorized origins/redirect URIs as required for Chrome Identity
- No client secret is used in the extension.

OAuth scopes (least privilege)

Prefer:

- `https://www.googleapis.com/auth/calendar.readonly`
  Or if available for FreeBusy:
- `https://www.googleapis.com/auth/calendar.freebusy` (if supported in your intended flow)

Use read-only if unsure; it's commonly accepted and works.

---

6) Data flow / messaging

Content → Background message

`type: "GET_AVAILABILITY"`
payload:

- rangePreset: `1w | 2w | custom`
- startDateISO (if custom)
- endDateISO (if custom)
- dailyStartTime: `"09:00"`
- dailyEndTime: `"17:00"`
- minDurationMinutes: number
- daysOfWeekEnabled: `[1..7]` (Mon=1..Sun=7)
- sourceTimeZone: `"America/Chicago"`
- outputTimeZone: `"America/New_York"`

Background response

`type: "AVAILABILITY_RESULT"`
payload:

- formattedText: string
- debug optional: computed blocks

---

7) Calendar API usage (FreeBusy)

Endpoint

`POST https://www.googleapis.com/calendar/v3/freeBusy`

Request body

- `timeMin`, `timeMax` (RFC3339 timestamps)
- `timeZone` (source timezone)
- `items: [{ id: "primary" }]`

Response contains

Busy intervals for the calendar(s), as RFC3339 timestamps.

---

8) Availability algorithm (core logic)

Inputs per day

- Daily working window: [dayStart, dayEnd] in **source timezone**
- Busy intervals from FreeBusy (in absolute timestamps)
- Min duration in minutes
- Enabled days of week

Steps

1. Build list of days in requested date range.
2. For each day:
    - If day-of-week not enabled → skip
    - Construct `windowStart` and `windowEnd` for that date in source timezone.
3. Convert busy intervals to overlaps with that day window:
    - For each busy interval, compute intersection with [windowStart, windowEnd]
    - Keep only overlapping intervals
4. Merge overlapping busy intervals (sort by start, then merge).
5. Subtract merged busy intervals from the daily window to get free blocks:
    - Start with cursor = windowStart
    - For each busy block:
        - if busy.start > cursor → free = [cursor, busy.start]
        - cursor = max(cursor, busy.end)
    - After loop:
        - if cursor < windowEnd → free = [cursor, windowEnd]
6. Filter free blocks < minDuration
7. Convert remaining free blocks to **output timezone** for display
8. Format into lines grouped by day.

Output choice (recommended for recruiting)

Return **free blocks**, not every discrete meeting slot start time.

---

9) Timezone handling requirements

User-specified timezones

- `sourceTimeZone`: timezone where they define working hours (and interpret "9am–5pm").

- `outputTimeZone`: timezone to display in email.

Implementation requirement

Use a timezone-aware date library in the extension for reliable conversion:

- Recommended: `luxon`
    - Bundle luxon into extension (no CDN)
    - Use `DateTime` with `.setZone(...)`

Conversion rules

- Construct day windows in source TZ.
- Treat FreeBusy timestamps as absolute instants; convert to source TZ for overlap computations.
- For display, convert final free blocks to output TZ, and format in output TZ.

---

10) Gmail integration details (button + insertion)

Compose detection strategy

In `content_script.js`:

- Use a `MutationObserver` on `document.body`.
- When nodes are added, scan for Gmail compose windows.
- For each compose window:
    - Find compose toolbar container (Gmail's formatting toolbar area)
    - Inject a button element if not already injected for that compose instance

Multi-compose support

- Gmail can have multiple compose windows open
- Tag each compose element with a custom attribute like:
    - `data-availability-button-injected="true"`
- Button click should target the correct compose editor (closest compose DOM ancestor)

Inserting text into Gmail body

- Find the editable div (usually `div[contenteditable="true"]` within compose)
- Insert at cursor if possible:

    - Focus the editor
    - Use `document.execCommand('insertText', false, text)` (deprecated but still widely works)
    - Fallback: insert HTML with `<br>` line breaks using `insertHTML`
- Always provide fallback: copy to clipboard if insertion fails

---

11) UI implementation (modal)

Modal requirements

- Inject a `<div id="availability-modal-root">` overlay into page.
- Modal fields listed above.
- Basic validation:
    - source timezone and output timezone non-empty
    - start < end
    - date range valid
    - min duration positive
- Buttons:
    - “Generate & Insert”
    - “Cancel”

No storage

- Values exist only while modal is open.
- If user closes modal, inputs reset.

---

12) Error handling (must implement)

Common errors

- OAuth denied → show friendly message (“Permission required…”)
- Token expired → re-auth
- Calendar API fails → show error + copy-to-clipboard fallback
- No free time found → insert message like:
    - “I’m fairly booked the next two weeks—could you share a few times that work for you?”

---

13) Implementation details per file

`src/content/content_script.js`

- Entry point
- Set up MutationObserver
- When compose detected, call `injectButton(composeEl)`
- On button click, open modal via `modal_ui.js`
- On modal submit:
  - send message to background: `GET_AVAILABILITY`
  - await response
  - call `insert_text.js` to insert into correct compose editor

`src/content/gmail_dom.js`

- Functions:
  - `findComposeWindows()`
  - `getComposeToolbar(composeEl)`
  - `getComposeEditor(composeEl)`
  - `isComposeDraftOpen(composeEl)`
- Responsible for Gmail DOM selectors and resilient searching.

`src/content/modal_ui.js` + `modal_ui.css`

- Render modal HTML
- Collect inputs
- Provide callbacks: `onSubmit(payload)`, `onCancel()`

`src/content/insert_text.js`

- `insertTextIntoCompose(composeEl, text)`
- Fallback path:
  - show a small toast “Copied to clipboard”
  - `navigator.clipboard.writeText(text)`

`src/background/service_worker.js`

- Listen for messages from content script
- On `GET_AVAILABILITY`:

    - ensure auth token via `google_auth.js`
    - call freebusy via `calendar_api.js`
    - compute availability via `availability.js`
    - format via `format.js`
    - respond with `{ formattedText }`

`src/background/google_auth.js`

- `getAccessToken(interactive = true)`
    - uses `chrome.identity.getAuthToken`
- If token invalid:
    - remove cached token and retry

`src/background/calendar_api.js`

- `fetchFreeBusy({ timeMin, timeMax, timeZone, token })`
- Return busy intervals for primary calendar

`src/background/time_utils.js`

- Luxon helpers:
    - parse date range
    - build per-day windows in source TZ
    - convert intervals between zones
    - format times

`src/background/availability.js`

- Implements algorithm:
    - merge busy intervals
    - subtract to compute free blocks
    - filter by min duration
- Returns structure:
    - `{ dayISO: [{ startISO, endISO }, ...] }` but in absolute instants or in output TZ depending on your design

`src/background/format.js`

- Convert computed free blocks into email-friendly text
- Group by day
- Include output timezone abbreviation if possible
- Keep formatting concise

`src/shared/constants.js`

- Defaults:
  - presets (1w, 2w)
  - min duration options
  - weekday mapping
  - max blocks per day (optional; e.g., show top 2 blocks to keep email short)

---

14) Development & testing checklist

Local test steps

1. Load extension unpacked in Chrome
2. Open Gmail
3. Compose new email
4. Ensure button appears
5. Click button, fill inputs
6. OAuth prompt appears (first run)
7. Inserted text appears in draft
8. Validate timezone conversion by testing different source/output TZ

Edge cases to test

- Multiple compose windows open
- Reply/forward compose vs new compose
- All-day events
- Busy blocks overlapping working window edges
- Daylight saving time boundary (Luxon should handle)

---

15) Notes on “no storage”

- Do NOT persist user settings.
- Do NOT store timezone preferences.
- Every button click → user fills the modal again.
- OAuth token caching will still happen internally via Chrome identity mechanisms (that’s fine and expected).
