Others

- Quotation Request Form (QRF) - broken down
- Step-by-step guide to using the OpenSTAAD API from Python
- Monetary / financial calculations inside your QRF → BOQ automation
- BIM/IFC models for QTO
- GHL
    - Integration vCX with GHL
- n8n ↔ Zendesk Web Widget (Classic) – JWT Integration
- MCP -> wiring diagram

Quotation Request Form (QRF) - broken down

Key inputs that must be supplied in a **Quotation Request Form (QRF)** for a Pre-Engineered Building (PEB) project are grouped below. If any of these items are missing the supplier cannot price or design the building accurately.

1. General project data
   - Project / building name
   - Exact site address (affects wind & seismic codes, freight, taxes)
   - Intended use / occupancy category
   - Required delivery / erection schedule
2. Geometry
   - Clear span width (outside-to-outside of main frames)
   - Building length (number of bays × bay spacing)
   - Eave height (bottom of knee connection to finished floor)
   - Roof slope (e.g., 1:10 or 2:12)
   - Required roof live load (kN/m² or psf)
   - Collateral loads (false ceiling, sprinklers, HVAC, lighting, etc.)
   - Floor live load (if mezzanine is requested)
3. Cladding specification
   - Roof sheeting: single-skin or insulated sandwich panel, thickness, color coating, finish
   - Wall sheeting: same detail as roof, plus liner panel if required
   - Skylight / daylight panels area (if any)
4. Environmental / code parameters
   - Basic wind speed (3-sec gust) or wind pressure
   - Seismic zone / design acceleration
   - Snow load (if applicable)
   - Temperature range for insulation calculation
   - Local building code and edition (IBC, Eurocode, IS 875, ASCE 7, etc.)
5. Structural add-ons
   - Overhead crane: capacity, hook height, class of usage, runway length
   - Mezzanine: area, floor loading, column grid
   - Canopies / lean-tos: width, projection, height
   - Parapet height, fascias, gutters, downpipes
6. Openings & accessories
   - Ridge vent or turbo vent area
   - Roll-up doors: clear opening width × height, quantity
   - Personnel doors: size, location, fire rating
   - Louvers / windows: size and quantity

    - Insulation: roof and/or wall R-value or U-value target
7. Architectural finishes & extras
    - Color chart reference for roof & wall
    - Special coatings (e.g., food-grade, coastal environment)
    - Fire-proofing requirements
    - Internal partitions (if supplied by PEB vendor)
8. Site constraints
    - Maximum truck length allowed to site
    - Crane reach restrictions for unloading / erection
    - Local welding restrictions or pre-approved vendors
9. Commercial terms
    - Scope split (supply-only vs. supply-and-erect)
    - Applicable taxes, duties, freight responsibility
    - Incoterms (EXW, FOB, CIF, DDP, etc.)
    - Payment milestones
    - Performance bond / insurance requirements

Supplying all of the above in the QRF ensures the PEB supplier returns an accurate quotation, preliminary general-arrangement drawing, and a detailed BOQ without back-and-forth clarifications.

Step-by-step guide to using the OpenSTAAD API from Python

1. Install prerequisites

```
pip install comtypes pywin32 openstaad
```

2. Launch STAAD.Pro and connect

```
import subprocess, time, comtypes.client
from pythoncom import CoInitialize, CoUninitialize

CoInitialize()                       # Initialise COM
staad_path = r"C:\Program Files\Bentley\Engineering\STAAD.Pro 2024\STAAD\Bentley.Staad.exe"
subprocess.Popen([staad_path])
time.sleep(8)                        # Wait for STAAD to open

openstaad = comtypes.client.GetActiveObject("StaadPro.OpenSTAAD")
```

3. Create or open a model

```
from pathlib import Path
std_file_path = Path.cwd() / "my_model.std"
length_unit = 4   # 4 = metres
force_unit   = 5  # 5 = kN
openstaad.NewSTAADFile(str(std_file_path), length_unit, force_unit)
time.sleep(3)
```

4. Define material and section

```
prop = openstaad.Property
prop.SetMaterialName("STEEL")

# European IPE200 section
prop_no = prop.CreateBeamPropertyFromTable(
    country_code=7,        # 7 = European database
    section_name="IPE200",
    type_spec=0,           # single section from table
    add_spec_1=0.0,
    add_spec_2=0.0
)
```

5. Add nodes and beams

```
geom = openstaad.Geometry
geom.CreateNode(1, 0, 0, 0)
geom.CreateNode(2, 5, 0, 0)
geom.CreateBeam(1, 1, 2)          # Beam 1: node 1 → node 2
prop.AssignBeamProperty(1, prop_no)
```

6. Supports and loads

```
sup = openstaad.Support
sup_no = sup.CreateSupportFixed()
sup.AssignSupportToNode(1, sup_no)
sup.AssignSupportToNode(2, sup_no)

ld = openstaad.Load
case = ld.CreateNewPrimaryLoad("Self-Weight")
ld.SetLoadActive(case)
ld.AddSelfWeightInXYZ(case, -1.0)   # factor −1 in global Y
```

7. Run the analysis (silent mode)

```
cmd = openstaad.Command
cmd.PerformAnalysis(6)      # 6 = static analysis
openstaad.SetSilentMode(1)
openstaad.Analyze()
while openstaad.isAnalyzing():
    time.sleep(2)
```

8. Retrieve results

```
from openstaad import Output
out = Output()
fx, fy, fz, mx, my, mz = out.GetMemberEndForces(beam=1, start=True, lc=1)
print("Start-end forces:", fx, fy, fz, mx, my, mz)
```

9. Clean up

```
openstaad.SaveModel(1)
CoUninitialize()
```

10. Helper wrappers

If you prefer a higher-level interface, install **OpenStaadPython**:

```
pip install openstaad
```

Then use convenience classes:

```
from openstaad import Geometry, Root
print(Geometry().GetBeamList())
print(Root().GetSTAADFile())
```

(Note: `openstaad` currently focuses on **querying** an **already-open** model.)

Documentation & Community

- **Official docs**:
  `C:\Program Files\Bentley\Engineering\STAAD.Pro 2024\OSAPP_Help`
  (Examples are mainly VB/C++; Python help lives in the Bentley forums.)
- **GitHub samples**:
  viktor-platform/sample-staad-integration

You can now create, modify, analyse and extract results from STAAD.Pro entirely from Python scripts.

Monetary / financial calculations inside your QRF → BOQ automation

For **monetary / financial calculations inside your QRF → BOQ automation**, you need two things:

1. **Exact decimal precision** (no binary-float rounding surprises).
2. **Convenience helpers** for currency formatting, FX, amortisation, etc.

---

1. Core precision → `decimal`

```
from decimal import Decimal, ROUND_HALF_UP


qty   = Decimal('12.50')     # kg
rate  = Decimal('78.35')     # $/kg
total = (qty * rate).quantize(Decimal('0.01'), ROUND_HALF_UP)
```

2. Money wrapper → `money` or `py-moneyed`

```
from money import Money


unit_price = Money('78.35', 'USD')
line_total = Money('12.50', 'USD') * unit_price
```

3. Excel / reporting → `openpyxl`, `xlsxwriter` (they both **preserve `Decimal` precision** when you write values).
4. Optional extras
   - `forex-python` – real-time FX rates for multi-currency bids.
   - `numpy-financial` – NPV, IRR, loan amortisation if you need financing tables.
   - `babel` – locale-aware currency formatting.

Quick recipe (fits your Python stack):

```
from decimal import Decimal
from openpyxl import Workbook
```

```
from money import Money

wb = Workbook()
ws = wb.active
ws.append(['Item', 'Qty (kg)', 'Rate ($)', 'Amount ($)'])

for item, qty, rate in [
        ('Column UC203', Decimal('253.4'), Decimal('1.08')),
        ('Beam UB305', Decimal('417.9'), Decimal('1.08'))]:
    amount = Money(qty * rate, 'USD')
    ws.append([item, float(qty), float(rate), str(amount)])

wb.save('BOQ_financial.xlsx')
```

All values stay exact (no rounding errors), and the worksheet shows standard $-formatting.

BIM/IFC models for QTO

Below is a concise, **Python-first** recipe that shows exactly how to plug an **IFC model** (provided with the QRF) into your **Quantity-Take-Off (QTO)** pipeline. It follows the **external-mode** pattern described in the BIM literature: the IFC file is **only a data source**; all logic runs in Python.

---

1. Install the core IFC library

```
pip install ifcopenshell pandas openpyxl
```

2. Load the IFC file and list every structural element

```
import ifcopenshell, ifcopenshell.util.element as util
from pathlib import Path

model = ifcopenshell.open(Path("rfx_structural.ifc"))

# Example: grab every IfcBeam, IfcColumn, IfcMember, etc.
elements = (model.by_type("IfcBeam") +
            model.by_type("IfcColumn") +
            model.by_type("IfcMember"))
```

3. Extract the quantities you need

```
rows = []
for e in elements:
    # 1. Identity
    name   = e.Name or e.GlobalId
    ifc_ent= e.is_a()
```

```
    # 2. Geometry quantities (IfcElementQuantity)
    qs = util.get_psets(e).get("Pset_ElementQuantity", {})
    length_m = float(qs.get("Length", 0))
    weight_kg= float(qs.get("Weight", 0))     # if the modeller exported it
    area_m2  = float(qs.get("SurfaceArea", 0))

    # 3. Material grade (IfcMaterial)
    mat = util.get_material(e)
    grade = mat.Name if mat else "Unknown"

    rows.append({
        "Item"      : name,
        "Type"      : ifc_ent,
        "Material"  : grade,
        "Length_m"  : length_m,
        "Weight_kg" : weight_kg,
        "Area_m2"   : area_m2
    })
```

4. Build a Pandas DataFrame → instant QTO table

```
import pandas as pd

df = pd.DataFrame(rows)
# Aggregate identical sections
qto = (df
       .groupby(["Type", "Material"], as_index=False)
       .agg({"Length_m":"sum",
             "Weight_kg":"sum",
             "Area_m2":"sum"}))
```

5. Export to Excel (ready for BOQ merge)

```
qto.to_excel("IFC_QTO.xlsx", index=False)
```

6. Optional: validate IFC quality first

Use **BIMvision** or **Solibri Anywhere** (free viewers) to visually inspect the model and confirm that all **Pset_ElementQuantity** properties are populated.

7. Handling federated models (multiple IFC files)

If the QRF supplies **several partial IFC files**, merge them once:

- **BIMvision → IFC Merge plugin** (permanent merge), or
- **IfcOpenShell** (memory merge) if you want to stay in Python.

8. When IFC lacks quantities

If the IFC only has geometry, compute **volume/length/area** yourself:

```
import ifcopenshell.geom as geom
settings = geom.settings()
shape = geom.create_shape(settings, e)
volume = shape.geometry.volume
```

9. Keep the workflow **MVD-compliant**

The NBIMS QTO guide recommends exporting with the **CDB-2010 MVD** view (or later). Ask the designer to tick that option in Revit/ArchiCAD so that **Pset_ElementQuantity** and **IfcMaterial** are automatically embedded.

Summary of the integration pattern

Step	Tool / Library	Purpose
IFC ingestion	`IfcOpenShell`	Parse geometry & properties
Quantity extraction	`IfcElementQuantity` or auto-calc	Length, area, volume, weight
Data shaping	`Pandas`	Group, sum, clean
Output	`openpyxl`	Excel BOQ ready for pricing

Step	Tool / Library	Purpose
Visual QC	BIMvision / Solibri	Confirm model quality

This external-mode approach keeps your Python stack lightweight, avoids STAAD-Pro geometry duplication, and produces **code-compliant QTO tables** in minutes

GHL

GHL

Integration vCX with GHL

Recommended Architecture

1. **Push vCX Conversations into GHL**

Use the https://highlevel.stoplight.io/docs/integrations/0443d7d1a4bd0-overview to:

- Create or update a **Contact** using `clientId` as the unique identifier
- Create a **Conversation** under that contact
- Add each message as a **Message** object inside the conversation

> All of this is supported via `POST /conversations/{id}/messages` and related endpoints .

2. **Pull GHL Replies into vCX**

Set up a **webhook subscription** in GHL to listen for:

- `message.incoming`
- `conversation.updated`

These webhooks will fire whenever a GHL user (or bot) replies. You can map the GHL `conversationId` to your `conversationId` using metadata or a lookup table.

> OAuth 2.0 is now required for all new integrations, so you'll need to register your app in GHL's Developer Portal .

Key GHL API Docs You'll Need

Table
Copy

Task	Endpoint	Notes
Create/Update Contact	`POST /contacts`	Use `clientId` as external ID
Create Conversation	`POST /conversations`	Link to contact

Task	Endpoint	Notes
Send Message	`POST /conversations/{id}/messages`	Includes text, type, timestamp
Listen for Replies	Webhook: `message.incoming`	Use to sync back to vCX

All endpoints are documented at:

📖 https://highlevel.stoplight.io/docs/integrations

vCX ↔ GoHighLevel – Two-Way Chat Sync (Node.js)

0. Prerequisites

- Node ≥ 18
- A GHL developer account (https://developers.gohighlevel.com)
- Your app registered in the portal with scopes: `contacts.write conversations.write conversations.read locations.read`
- Redirect URI set to `https://yourdomain.com/auth/callback`
- Environment variables:

```
GHL_CLIENT_ID=xxxxxxxx
GHL_CLIENT_SECRET=xxxxxxxx
GHL_REDIRECT_URI=https://yourdomain.com/auth/callback
```

1. Install dependencies

`package.json (excerpt)`

```
{
  "type": "module",
  "dependencies": {
    "axios": "^1.6.0",
    "dotenv": "^16.3.1",
    "express": "^4.18.2"
```

```
  }
}
```

```
npm install
```

2. Minimal Express server skeleton

```
server.js (top)
```

```
import 'dotenv/config';
import express from 'express';
import axios from 'axios';
import crypto from 'crypto';
const app = express();
app.use(express.json());

const PORT = process.env.PORT || 3000;

/* --- In-memory maps for demo purposes --- */
const tokenStore = new Map();          // locationId -> {access_token, refresh_token, expires_
const conversationMap = new Map();     // vcxConversationId -> ghlConversationId

app.listen(PORT, () => console.log(`Listening on :${PORT}`));
```

3. OAuth 2.0 – Authorization URL

```
GET /install
```

```
app.get('/install', (req, res) => {
  const state = crypto.randomUUID();
  const url = `https://marketplace.gohighlevel.com/oauth/chooselocation?response_type=code&cli
  res.redirect(url);
});
```

4. OAuth 2.0 – Exchange code for tokens

```
GET /auth/callback
```

```
app.get('/auth/callback', async (req, res) => {
  const { code, locationId } = req.query;
  const { data } = await axios.post('https://services.leadconnectorhq.com/oauth/token', {
    client_id: process.env.GHL_CLIENT_ID,
    client_secret: process.env.GHL_CLIENT_SECRET,
```

```
    grant_type: 'authorization_code',
    code,
    redirect_uri: process.env.GHL_REDIRECT_URI
  });
  tokenStore.set(locationId, {
    access_token: data.access_token,
    refresh_token: data.refresh_token,
    expires_at: Date.now() + data.expires_in * 1000
  });
  res.send(`Sub-account ${locationId} connected.`);
});
```

5. Helper – get valid access token (auto-refresh)

```
async function getToken(locationId) {
  let t = tokenStore.get(locationId);
  if (!t) throw new Error('Location not authorized');

  if (Date.now() > t.expires_at - 60_000) {
    const { data } = await axios.post('https://services.leadconnectorhq.com/oauth/token', {
      client_id: process.env.GHL_CLIENT_ID,
      client_secret: process.env.GHL_CLIENT_SECRET,
      grant_type: 'refresh_token',
      refresh_token: t.refresh_token
    });
    t = {
      access_token: data.access_token,
      refresh_token: data.refresh_token,
      expires_at: Date.now() + data.expires_in * 1000
    };
    tokenStore.set(locationId, t);
  }
  return t.access_token;
}
```

6. Upsert Contact (by vCX clientId)

```
POST /contact
```

```
app.post('/contact', async (req, res) => {
  const { locationId, clientId, email, phone, name } = req.body;
  const token = await getToken(locationId);

  // Search by externalId first
  const search = await axios.get(`https://rest.gohighlevel.com/v1/contacts/?query=${clientId}&
    headers: { Authorization: `Bearer ${token}` }
  });
```

```
  let contactId = search.data.contacts?.[0]?.id;

  if (!contactId) {
    const { data } = await axios.post('https://rest.gohighlevel.com/v1/contacts/', {
      locationId,
      name,
      email,
      phone,
      source: 'vCX',
      tags: ['vCX'],
      customFields: [{ id: 'clientId', value: clientId }]
    }, { headers: { Authorization: `Bearer ${token}` } });
    contactId = data.contact.id;
  }
  res.json({ contactId });
});
```

7. Create Conversation & Push Messages

```
POST /push-message
```

```
app.post('/push-message', async (req, res) => {
  const { locationId, clientId, vcxConversationId, fromUser, body, timestamp } = req.body;
  const token = await getToken(locationId);

  // 1. Ensure contact
  const { contactId } = (await axios.post(`http://localhost:${PORT}/contact`, {
    locationId, clientId, email: `${clientId}@example.com`, name: clientId
  })).data;

  // 2. Ensure conversation
  let ghlConvId = conversationMap.get(vcxConversationId);
  if (!ghlConvId) {
    const { data } = await axios.post('https://rest.gohighlevel.com/v1/conversations/', {
      locationId,
      contactId,
      type: 'chat'
    }, { headers: { Authorization: `Bearer ${token}` } });
    ghlConvId = data.conversation.id;
    conversationMap.set(vcxConversationId, ghlConvId);
  }

  // 3. Push message
  await axios.post(`https://rest.gohighlevel.com/v1/conversations/${ghlConvId}/messages`, {
    type: fromUser ? 'Inbound' : 'Outbound',
    message: body,
    dateAdded: new Date(timestamp).toISOString()
  }, { headers: { Authorization: `Bearer ${token}` } });
```

```
  res.sendStatus(200);
});
```

8. Receive GHL Replies via Webhook

```
POST /webhook
```

```
app.post('/webhook', (req, res) => {
  const { type, locationId, conversationId, message } = req.body;

  if (type !== 'message.incoming') return res.sendStatus(200);

  // Reverse lookup
  let vcxConvId;
  for (const [vId, gId] of conversationMap.entries()) {
    if (gId === conversationId) vcxConvId = vId;
  }
  if (!vcxConvId) return res.sendStatus(200);

  // TODO: forward to vCX backend
  console.log('Forward to vCX:', { vcxConversationId: vcxConvId, fromUser: false, body: messag

  res.sendStatus(200);
});
```

Register this URL in GHL → Settings → API → Webhooks.

9. Quick test with cURL

```
# 1. Start your server
node server.js

# 2. Install the app (open in browser)
open http://localhost:3000/install

# 3. Push a message
curl -X POST http://localhost:3000/push-message \
  -H "Content-Type: application/json" \
  -d '{"locationId":"LOC_ID","clientId":"c_abc123","vcxConversationId":"conv_456","fromUser":t
```

10. Production checklist

- Use persistent storage (Redis/Postgres) instead of in-memory maps.
- Verify webhook signatures (GHL sends headers `X-Signature`).

- Rate-limit token refresh.
- Handle pagination when searching contacts.
- Wrap axios calls in retries with exponential backoff.

Last updated 2024-07-20

n8n ↔ Zendesk Web Widget (Classic) – JWT Integration

Internal Documentation v1.0 – 2025-08-25

> **Goal**
> Allow visitors authenticated through your n8n chat front-end to start a Zendesk Web Widget session, while still requiring a human agent to approve any ticket creation.

1. Prerequisites

Item	Where to find
Zendesk account with **Web Widget (Classic)** enabled	Admin Center → Channels → Widget
Shared Secret for JWT	Admin Center → Channels → Chat → Widget → *Authentication*
n8n instance reachable from the public internet	`https://your-n8n.com`
Existing n8n workflow that pauses for human review	(Human-in-the-Loop)

2. High-Level Flow

1. Visitor loads your web-chat.
2. Front-end requests a **JWT** from n8n.
3. n8n signs and returns the JWT.
4. Zendesk Web Widget starts an **authenticated chat** session.
5. **Human-in-the-Loop** still controls *ticket creation* (Wait node).

---

3. n8n Endpoints

3.1 JWT Issuer (`POST /webhook/zendesk-jwt`)

Purpose: Zendesk will call this endpoint to verify the visitor.

Workflow Steps

Node	Settings
Webhook	Path = `/webhook/zendesk-jwt` (POST)
Lookup User	Any node that confirms the `user_token` sent by Zendesk is valid (Database, Google Sheets, etc.).
JWT Sign	Algorithm = `HS256`
Respond to Webhook	Status = `200`

Example cURL

```
curl -X POST \
  'https://your-n8n.com/webhook/zendesk-jwt?user_token=abc123' \
  -H 'Content-Type: application/json'
```

Expect:

```
{"jwt":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."}
```

3.2 Optional Token Generator for Front-End (`GET /webhook/chat-token`)

If your front-end needs to fetch the token itself (instead of letting Zendesk call the endpoint directly), create a second simple workflow:

Node	Settings
Webhook	Method = `GET`

Node	Settings
JWT Sign	Same payload & secret as above
Respond to Webhook	Body = `{"jwt":"{{ $('JWT Sign').item.jwt }}"}`

---

4. Zendesk Configuration

Admin Center → Channels → **Chat** → **Widget** → **Authentication**

Field	Value
Authentication Method	JWT
JWT URL	`https://your-n8n.com/webhook/zendesk-jwt`
JWT Secret	*Paste the same Shared Secret used in n8n*

5. Front-End Snippet

Add this after the Zendesk Web Widget script is loaded:

```
<script>
  // Replace with your n8n endpoint if you created /webhook/chat-token
  fetch('/api/n8n/get-chat-token', { credentials: 'include' })
    .then(r => r.json())
    .then(({ jwt }) => {
      zE('webWidget', 'chat:setJwtFn', callback => callback(jwt));
    });
</script>
```

> If you let Zendesk call your endpoint directly, omit the fetch and simply use the JWT URL configured above.

6. Human-in-the-Loop Remains Intact

Your existing workflow already has a **Wait** node that pauses until an agent approves.
Nothing in the JWT flow changes that—ticket creation only proceeds **after** the webhook resume call.

7. Security Checklist

- [ ] HTTPS only (n8n & Zendesk endpoints).
- [ ] Rotate Shared Secret periodically → update both Zendesk and n8n JWT node.
- [ ] Log every JWT issuance (`iat`, IP, user_token).
- [ ] Validate `user_token` strictly; deny unknown tokens immediately.

8. Troubleshooting Quick-Table

Symptom	Likely Cause	Fix
Widget shows “Unable to authenticate”	Wrong Shared Secret or expired `iat`	Check secret & timestamp
404 from Zendesk	Wrong JWT URL	Ensure `https://your-n8n.com/webhook/zendesk-jwt` is publicly reachable
“Invalid JWT format”	Payload missing required claims	Ensure `name`, `email`, `jti`, `iat` are present

9. Change Log

Date	Author	Notes
2025-08-25	DevOps	Initial draft based on Zendesk article 4408838925082

MCP -> wiring diagram

Below is a minimal but complete “wiring diagram” + code snippets that let:

- a React/JS chat UI
- talk to your existing backend over the WebSocket `wss://backend.chatbuilder.com/events/listen`
- which forwards every user sentence to **your** Node orchestrator (the “bot”)
- that hosts an LLM (Anthropic or OpenAI) **and** an MCP client
- which calls an MCP **server** (also Node) that owns the Airtable CRUD helpers
- and finally ships the answer back the same chain.

No Claude Desktop, no stdio, everything is plain HTTP/SSE inside your own VPC.

---

1. Component map

Chat UI ⇄ WSS ⇄ Backend.chatbuilder.com ⇄ HTTP ⇄ Bot/orchestrator ⇄ SSE ⇄ Airtable-MCP-server
(React) (existing) (your Node service) (your Node MCP server)

- The **bot** keeps the LLM API key and the MCP client.
- The **MCP server** only knows Airtable PAT + base ID and exports tools like `airtable:select_records`, `airtable:create_record`, …
- Both services are Dockerised and scale horizontally.

---

2. Airtable MCP server (Node, SSE transport)

Install
mkdir airtable-mcp && cd airtable-mcp
npm init -y
npm install @modelcontextprotocol/sdk airtable dotenv
server.js
JavaScript
Copy

```
import { Server } from "@modelcontextprotocol/sdk/server/index.js";
import { SSEServerTransport } from "@modelcontextprotocol/sdk/server/sse.js";
import express from "express";
import Airtable from "airtable";
import "dotenv/config";
```

```
const app = express();
app.use(express.json());

const port = process.env.PORT || 8001;
const base = new Airtable({apiKey: process.env.AIRTABLE_PAT})
               .base(process.env.AIRTABLE_BASE_ID);

// 1. describe tools
const tools = [
  {
    name: "airtable:select_records",
    description: "List records from a table",
    inputSchema: {
      type: "object",
      properties: {
        table:   { type: "string" },
        filter:  { type: "string" },
        maxRecords: { type: "number", default: 10 }
      },
      required: ["table"]
    }
  },
  {
    name: "airtable:create_record",
    description: "Insert one record",
    inputSchema: {
      type: "object",
      properties: {
        table: { type: "string" },
        fields: { type: "object" }
      },
      required: ["table", "fields"]
    }
  }
];

// 2. instantiate MCP server
const server = new Server(
  { name: "airtable-mcp", version: "1.0.0" },
  { capabilities: { tools: {} } }
```

```
);

server.setRequestHandler("tools/list", async () => ({ tools }));
server.setRequestHandler("tools/call", async (req) => {
  const { name, arguments: args } = req.params;
  if (name === "airtable:select_records") {
    const recs = await base(args.table)
      .select({ maxRecords: args.maxRecords || 10, filterByFormula: args.filter || "" })
      .all();
    return {
      records: recs.map(r => ({ id: r.id, fields: r.fields }))
    };
  }
  if (name === "airtable:create_record") {
    const created = await base(args.table).create([{ fields: args.fields }]);
    return { id: created[0].id };
  }
  throw new Error("Unknown tool");
});

// 3. expose SSE endpoints
app.get("/sse", async (req, res) => {
  const transport = new SSEServerTransport("/message", res);
  await server.connect(transport);
});

app.post("/message", (req, res) => {
  const transport = SSEServerTransport.get(req.query.sessionId);
  if (transport) transport.handlePostMessage(req, res);
});

app.listen(port, () => console.log(`Airtable MCP listening on :${port}`));
```

.env
Copy

```
AIRTABLE_PAT=patXXXXXXXXXX
AIRTABLE_BASE_ID=appXXXXXXXXXX
```

Run

node server.js → http://localhost:8001/sse (SSE endpoint)

---

3. Bot/orchestrator (Node, hosts LLM + MCP client)

---

mkdir bot && cd bot
npm init -y
npm install @modelcontextprotocol/sdk axios dotenv express
bot.js
JavaScript
Copy

```
import { Client } from "@modelcontextprotocol/sdk/client/index.js";
import { SSEClientTransport } from "@modelcontextprotocol/sdk/client/sse.js";
import axios from "axios";
import express from "express";
import "dotenv/config";

const app = express();
app.use(express.json());

// 1. connect MCP client to airtable server
const mcp = new Client({ name: "chat-bot", version: "1.0.0" });
const transport = new SSEClientTransport("http://localhost:8001/sse");
await mcp.connect(transport);
const tools = await mcp.listTools();

// 2. small helper: talk to LLM
async function callLLM(messages) {
  const body = {
    model: process.env.LLM_MODEL,        // "claude-3-5-sonnet-20241022" or "gpt-4-turbo"
    messages,
    tools: tools.map(t => t.inputSchema ? { ...t, function: t.inputSchema } : t),
    tool_choice: "auto",
    max_tokens: 2000
  };

  const url = process.env.LLM_PROVIDER === "anthropic"
    ? "https://api.anthropic.com/v1/messages"
    : "https://api.openai.com/v1/chat/completions";
```

```
  const headers = process.env.LLM_PROVIDER === "anthropic"
    ? { "x-api-key": process.env.ANTHROPIC_KEY, "content-type": "application/json" }
    : { "authorization": `Bearer ${process.env.OPENAI_KEY}`, "content-type":
"application/json" };

  const { data } = await axios.post(url, body, { headers });
  return data;     // returns Claude or OpenAI shape
}

// 3. single HTTP endpoint that backend.chatbuilder.com will call
app.post("/handle_turn", async (req, res) => {
  const userSentence = req.body.text;          // comes from backend via HTTP
  const conversation = [{ role: "user", content: userSentence }];

  // first LLM call
  let llmResp = await callLLM(conversation);
  let assistantMsg = llmResp.content || llmResp.choices[0].message;

  // handle tool calls
  if (assistantMsg.tool_calls || assistantMsg.function_call) {
    const toolCalls = assistantMsg.tool_calls || [assistantMsg.function_call];
    for (const tc of toolCalls) {
      const name = tc.function?.name || tc.name;
      const args = JSON.parse(tc.function?.arguments || tc.arguments);
      const result = await mcp.callTool(name, args);
      conversation.push(assistantMsg);
      conversation.push({ role: "tool", tool_call_id: tc.id, content: JSON.stringify(result)
});
    }
    // second call with tool results
    llmResp = await callLLM(conversation);
    assistantMsg = llmResp.content || llmResp.choices[0].message;
  }

  const replyText = assistantMsg.content || assistantMsg.text || assistantMsg;
  res.json({ reply: replyText });   // goes back to backend.chatbuilder.com
});

app.listen(3000, () => console.log("Bot/orchestrator on :3000"));
```

.env
Copy

```
LLM_PROVIDER=anthropic        # or openai
ANTHROPIC_KEY=sk-ant-xxx
OPENAI_KEY=sk-xxx
LLM_MODEL=claude-3-5-sonnet-20241022   # or gpt-4-turbo
```

---

4. Glue inside backend.chatbuilder.com

You already have a WebSocket handler.
Add (pseudo):
JavaScript
Copy

```
// when a message arrives from UI
ws.on('message', async (data) => {
  const { text, userId } = JSON.parse(data);
  // forward to bot/orchestrator
  const { data: { reply } } = await axios.post(
    "http://bot-service:3000/handle_turn",
    { text, userId }
  );
  // send answer back to same websocket
  ws.send(JSON.stringify({ type: "bot_reply", text: reply }));
});
```

---

5. One-shot docker-compose for local dev

---

yaml
Copy

```
version: "3.8"
services:
```

```
  airtable-mcp:
    build: ./airtable-mcp
    ports: ["8001:8001"]
    env_file: ./airtable-mcp/.env
  bot:
    build: ./bot
    ports: ["3000:3000"]
    env_file: ./bot/.env
    depends_on: [airtable-mcp]
```

`docker compose up` → everything spins up, UI talks to your existing backend, backend forwards to bot, bot calls Airtable via MCP, answer flows back.

---

6. What you gained

- Chat UI ⇄ WSS stays untouched.
- Backend.chatbuilder.com only needs to **forward** text to the bot service; no Airtable keys, no LLM keys, no MCP logic.
- Airtable CRUD lives in its own container; expose extra tools (update, delete, linked tables, …) by editing only the MCP server.
- Swap Anthropic ↔ OpenAI by changing one env var.
- Add Google-Calendar MCP server on port 8002, register it in the bot startup loop—zero other changes.