Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image
Uploaded image

Look at the images. i want to brain storm an idea with you. These are paper forms

I need a way to define these forms digitally so that I can do the following

1. Render them for user input
2. Store them in database
3. Make report out of their data

First, the idea is to have a domain specific language to define them

Perfect — I see what you're aiming for.
These PGCB forms (substation log sheets, duty rosters, inspection forms, etc.) are *highly tabular, repetitive, and structured*. A domain-specific language (DSL) makes sense, because:

1. You want to **digitize paper forms once**,
2. Then use the definition to **render UI for input**,
3. **Persist the captured values** in a database,
4. And later **generate reports/exports**.

Brainstorm: Designing a DSL for Forms

We need three layers in the DSL:

1. **Form Metadata** (title, reference number, effective date, sectioning).
2. **Form Structure** (sections, tables, fields).
3. **Field Properties** (type, validation, storage mapping).

Example DSL Structure (YAML/JSON based)

Let's take your **"Sub-Station Performance"** form as an example.

```yaml
form:
  id: substation_performance
  title: "Consolidated statement of Sub-Station & Transmission Line Performance"
  version: "1.0"
  metadata:
    organization: "Power Grid Company of Bangladesh Ltd."
```

```
      document_no: "QF-GMD-06"
      effective_date: "2007-02-15"
      revision_no: "00"
    sections:
      - id: substation_performance
        title: "A) Sub-Station Performance"
        table:
          columns:
            - name: "sl_no"
              label: "Sl. No."
              type: integer
            - name: "capacity_mva"
              label: "Total Sub-station capacity (MVA)"
              type: number
            - name: "forced_interruptions"
              label: "Number of interruption (Forced)"
              type: integer
            - name: "scheduled_interruptions"
              label: "Number of interruption (Scheduled)"
              type: integer
            - name: "total_interruptions"
              formula: "forced_interruptions + scheduled_interruptions"
              type: integer
            - name: "upto_30_min"
              label: "Number of interruption (Upto 30 minutes)"
              type: integer
            - name: "upto_1_hr"
              label: "Number of interruption (Upto 01 hour)"
              type: integer
            - name: "more_than_1_hr"
              label: "Number of interruption (More than 01 hour)"
              type: integer
            - name: "energy_mwh"
              label: "Amount of Energy Interruption (MkWh)"
              type: decimal
            - name: "remarks"
              label: "Remarks"
              type: text
      - id: line_performance
        title: "B) Line Performance"
        table: ...
```

Benefits of DSL

✅ Render for Input

- Web app can parse DSL → generate dynamic forms (React/Vue/Blazor).
- Field types (`integer`, `decimal`, `text`) → map to UI widgets.
- Constraints (like formulas, required fields) → validation.

✅ Store in Database

- DSL → defines schema (tables or JSONB in PostgreSQL).
- For tabular sections, each row can be stored as a record.

✅ Make Reports

- Since fields are explicitly defined, you can auto-generate tabular reports.
- Formulas in DSL (`total = forced + scheduled`) → calculated in reports.

DSL Design Options

1. **YAML/JSON DSL**
    - Easy to read/write.
    - Developers & even engineers can update without much coding.
2. **SQL-backed DSL**
    - Store forms in DB as JSON schema.
    - Easy dynamic queries.
3. **Text-based DSL** (like Markdown but extended)
   Example:

```
vbnet

[Form: Substation Performance]
Field: capacity_mva (Number, label="Sub-station capacity in MVA")
Field: forced_interruptions (Integer)
Field: scheduled_interruptions (Integer)
```

→ Parsed into structured data.

Next Step for Brainstorming

I suggest we prototype one **minimal DSL** and then build:

- **Renderer** (React/Blazor UI)
- **Storage mapping** (Postgres JSONB or normalized schema)
- **Reporting generator** (CSV/PDF export)

Would you like me to **propose a concrete DSL syntax** (e.g., YAML vs custom text DSL) and show how one of your PGCB forms (say the **Monthly Shift Duty Roster**) could be represented in it?

json or yaml seems good for dsl I think

Great—JSON or YAML is perfect. Here's a concrete, **practical DSL** you can start using today, plus examples of your PGCB forms and how they'd flow into UI, DB, and reports.

---

1) The DSL (YAML/JSON)

Look at the images. i want to brain storm an idea with you. These are paper forms I need a way to define these forms digitally so that I can do the following 1. Render them for user input 2. Store them in database 3. Make report out of their data First, the idea is to have a domain specific language to define them. 1
json or yaml seems good for dsl I think. 3
Lets add some rule - A grid can have finite rows, or inifinite rows Lets add the concept of Page in a Form. 11
Since we will run reports. I need store the data in table form. Lets discuss ideas. 14
Lets define the dsl first so that we can start somewhere. Can you write a DSL spec for the yaml. 19

We need to able to derive the table schema from yaml. Cna it be made generic. 28
based on all discussions above - can we design spec for a visual form designer. 31

Think of four layers:

- **form** → metadata + global options
- **sections[]** → groups the page into blocks
- **widgets** → input elements or tables inside sections
- **fields/columns** → actual data points (typed, validated, computed)

Core types

- `string | text | integer | decimal | date | time | datetime | bool | enum | attachment | signature`
- Hints: `unit`, `placeholder`, `readonly`, `required`, `pattern`, `min`, `max`
- **Formula**: simple expressions over same-row fields (`forced + scheduled`), or aggregations in a table (`sum(energy_mwh)`)

ID rules

- `form.id`, `section.id`, `widget.id`, `field/column.name` must be kebab/ snake-case; unique within scope.
- Stable IDs = stable DB columns.

A. DSL schema (concise)

```yaml
form:
  id: string
  title: string
  version: string
  locale: [ "en", "bn" ]             # optional
  labels?:                           # optional i18n
    en: string
    bn: string
  meta:                              # free-form header data you want printed
    document_no?: string
    effective_date?: date
    revision_no?: string
    organization?: string

  sections:
    - id: string
      title: string
      widgets:
        - type: "group" | "field" | "table" | "checklist" | "grid"
          id: string
          title?: string
          # 1) Single field (label-value)
          when?: "expression over other values"
          field?: { name, label, type, ...validation, ...ui }

          # 2) Group = multiple single fields arranged in a small layout
          fields?: [ { name, label, type, ... } ]
          layout?: { columns: 2 }         # renderer hint

          # 3) Table = list/rows with named columns
          table?:
            columns: [ { name, label, type, required?, unit?, enum?, formula?, pattern?, min?,
max?, widget? } ]
            row_init?: "n"                # empty rows to show initially
            row_generators?:             # create rows dynamically
              - type: "range"            # e.g., days 1..31, hours, etc.
                name: "day"
                from: 1
                to: 31
              - type: "times"
                name: "time"
                start: "07:00"
                end:   "22:00"
                step_minutes: 60
            aggregates?:                 # footer totals
              - name: "total_energy"
                label: "Total Energy (MkWh)"
                expr: "sum(energy_mwh)"

          # 4) Checklist = yes/no or status list
          checklist?:
            items:
              - key: "overall-cleanliness"
                label: "Overall cleanliness"
                type: "enum"
                enum: ["Clean","Not Cleaned"]

          # 5) Grid = 2D matrix (e.g., roster)
          grid?:
            rows:
              generator: { type: "names" }     # or explicit array
            columns:
              generator:
                type: "days-of-month"
            cell:
              type: "enum"
              enum: ["A","B","C","G","F","Ad",""]
              help: "A=06–14, B=14–22, C=22–06, G=Govt Holiday, F=Weekly Holiday, Ad=Additional Duty"

  storage:
    mode: "normalized" | "jsonb"      # how to materialize (see DB section)
```

Use JSON 1:1 with the same structure if you prefer.

---

2) Examples from your forms

Example 1 — “A) Sub-Station Performance” (table with computed column)

```yaml
form:
  id: substation-performance
  title: "Consolidated statement of Sub-Station & Transmission Line Performance"
  version: "1.0"
  meta:
    organization: "Power Grid Company of Bangladesh Ltd."
    document_no: "QF-GMD-06"
    effective_date: "2007-02-15"
    revision_no: "00"

  sections:
    - id: header
      title: "Header"
      widgets:
        - type: "group"
          id: "substation-identity"
          fields:
            - { name: substation, label: "Substation", type: string, required: true }
            - { name: substation_id, label: "Substation Identification No", type: string }
            - { name: reference_file, label: "Reference File", type: string }
            - { name: month, label: "Month", type: date }   # store first day of month

    - id: a-substation
      title: "A) Sub-Station Performance"
      widgets:
        - type: "table"
          id: "substation-perf"
          table:
            columns:
              - { name: sl_no, label: "Sl", type: integer }
              - { name: capacity_mva, label: "Total Sub-station capacity (MVA)", type: decimal,
min: 0 }
              - { name: forced, label: "Interruptions (Forced)", type: integer, min: 0 }
              - { name: scheduled, label: "Interruptions (Scheduled)", type: integer, min: 0 }
              - { name: total, label: "Total Interruptions", type: integer, formula: "forced +
scheduled" }
              - { name: upto_30_min, label: "Upto 30 minutes", type: integer, min: 0 }
              - { name: upto_1_hr, label: "Upto 01 hour", type: integer, min: 0 }
              - { name: more_than_1_hr, label: "More than 01 hour", type: integer, min: 0 }
              - { name: energy_mwh, label: "Energy Interruption (MkWh)", type: decimal, min: 0,
unit: "MkWh" }
              - { name: remarks, label: "Remarks", type: text }
            aggregates:
              - { name: sum_energy_mwh, label: "Total Energy (MkWh)", expr: "sum(energy_mwh)" }

    - id: b-line
      title: "B) Line Performance"
      widgets:
        - type: "table"
          id: "line-perf"
          table:
            columns:
              - { name: sl_no, label: "Sl", type: integer }
              - { name: length_ckm, label: "Total length of Transmission line (Ckt. Km.)", type:
decimal, min: 0 }
              - { name: no_of_sections, label: "No. of Section in Transmission line", type:
integer, min: 0 }
              - { name: forced, label: "Interruptions (Forced)", type: integer, min: 0 }
              - { name: scheduled, label: "Interruptions (Scheduled)", type: integer, min: 0 }
              - { name: total, label: "Total Interruptions", type: integer, formula: "forced +
scheduled" }
              - { name: upto_30_min, label: "Upto 30 minutes", type: integer, min: 0 }
              - { name: upto_1_hr, label: "Upto 01 hour", type: integer, min: 0 }
              - { name: more_than_1_hr, label: "More than 01 hour", type: integer, min: 0 }
              - { name: energy_mwh, label: "Energy Interruption (MkWh)", type: decimal, min: 0 }
              - { name: remarks, label: "Remarks", type: text }
```

Example 2 — Monthly Shift Duty Roster (grid over days)

```yaml
form:
  id: monthly-shift-duty-roster
  title: "Monthly Shift Duty Roster"
  version: "1.0"

  sections:
    - id: header
      title: "Header"
      widgets:
        - type: "group"
          id: "context"
          fields:
            - { name: grid_circle, label: "Grid Circle", type: string }
            - { name: gmd, label: "GMD", type: string }
            - { name: sub_station, label: "Sub-station", type: string, required: true }
            - { name: sub_station_id, label: "Sub-Station Identification No", type: string }
            - { name: month, label: "Month", type: date, required: true }
    - id: roster
      title: "Roster"
      widgets:
        - type: "grid"
          id: "roster-grid"
          grid:
            rows:
              generator: { type: "names" }     # UI lets you add names; or prefill from HR sync
            columns:
              generator:
                type: "days-of-month"          # derives column count from header.month
            cell:
              type: enum
              enum: ["A","B","C","G","F","Ad",""]
              help: "A=06–14, B=14–22, C=22–06, G=Govt Holiday, F=Weekly, Ad=Additional"
```

Example 3 — Transformer Log Sheet (hourly time series, two transformers)

```yaml
form:
  id: transformer-log-sheet
  title: "Grid Sub-Station Transformer Log Sheet"
  version: "1.0"

  sections:
    - id: context
      title: "Context"
      widgets:
        - type: "group"
          id: "ctx"
          fields:
            - { name: date, label: "Date", type: date, required: true }
            - { name: day_name, label: "Day", type: string }
            - { name: transformer_ids, label: "Transformers in Bay", type: string }

    - id: tr-a
      title: "Transformer TR #A"
      widgets:
        - type: "table"
          id: "tr-a-table"
          table:
            row_generators:
              - { type: "times", name: "time", start: "07:00", end: "22:00", step_minutes: 60 }
            columns:
              - { name: time, label: "Time", type: time, readonly: true }
              - { name: kv_bus_voltage, label: "kV bus voltage", type: decimal }
              - { name: pf_lv, label: "PF (LV)", type: decimal, min: -1, max: 1 }
              - { name: mw_lv, label: "MW (LV)", type: decimal }
              - { name: mvar_lv, label: "MVAR (LV)", type: decimal }
              - { name: amp_lv_r, label: "Ampere R (LV)", type: decimal }
              - { name: amp_lv_y, label: "Ampere Y (LV)", type: decimal }
              - { name: amp_lv_b, label: "Ampere B (LV)", type: decimal }
              - { name: winding_oil_temp, label: "Winding/Oil Temp", type: decimal, unit: "°C" }
              # …add OLTC, TAP, etc. as per sheet
            aggregates:
              - { name: max_oil_temp, label: "Max Oil Temp", expr: "max(winding_oil_temp)" }

    - id: signatures
      title: "Signatures"
      widgets:
        - type: field
          id: "sig-a"
          field: { name: sig_a_shift, label: "Signature (A shift)", type: signature }
        - type: field
```

```
          id: "sig-b"
          field: { name: sig_b_shift, label: "Signature (B shift)", type: signature }
```

---

3) Rendering: how UI uses the DSL

- **Frontend** (React/Blazor): parse DSL → render widgets:
  - **`field`/`group`**: standard form controls.
  - **`table`**: editable grid; support formulas & row add/remove.
  - **`grid`**: 2D matrix component optimized for rosters.
  - **`checklist`**: compact yes/no/status UI.
- **Validation**: enforce `required`, `min`/`max`, `pattern`, `enum`.
- **Computed**: recalc on change (`formula`, `aggregates`).
- **Internationalization**: use `labels.en/bn` if present.

---

4) Storage model (PostgreSQL)

You have two practical options; you can support both.

Option A — JSONB-first (fast to ship)

- `form_definitions(id, version, dsl_jsonb)`
- `form_instances(id, form_id, version, created_at, context -> jsonb, data -> jsonb)`
- Pros: flexible, evolving; perfect for varied forms.
- For reporting, **materialize views** or **jsonb_path_query** to extract columns.

Option B — Auto-normalized per table widget (best for heavy analytics)

- Keep JSONB instance for audit + re-render.
- Additionally, for each `table` widget, create a physical table:
  - `<form_id>__<widget_id>` with columns = `table.columns`.
  - Write rows there on submit.
- Provide a tiny **codegen** that turns DSL → `CREATE TABLE` + `INSERT/UPSERT`.

  Hybrid is common: JSONB for the whole submission + normalized tables only for the widgets you'll report on frequently (e.g., energy interruptions, roster).

---

5) Reporting

With Option B, your SQL is straightforward.

Example: monthly total energy interruption by substation

```
sql

SELECT i.context->>'substation' AS substation,
       date_trunc('month', (i.context->>'month')::date) AS month,
       SUM(t.energy_mwh) AS total_mkwh
FROM form_instances i
JOIN substation_performance__substation-perf t
  ON t.instance_id = i.id
WHERE i.form_id = 'substation-performance'
GROUP BY 1,2
ORDER BY 2,1;
```

If JSONB-only:

```
sql

SELECT i.context->>'substation' AS substation,
       date_trunc('month', (i.context->>'month')::date) AS month,
       SUM( (row->>'energy_mwh')::numeric ) AS total_mkwh
FROM form_instances i,
     LATERAL jsonb_array_elements(i.data->'sections'->'a-substation'->'substation-perf'->'rows')
AS row
WHERE i.form_id = 'substation-performance'
GROUP BY 1,2;
```

Roster coverage check (who worked C shift last week)

```
sql

SELECT name, day
FROM monthly_shift_duty_roster__roster-grid
WHERE cell_value = 'C'
  AND day BETWEEN DATE '2025-09-01' AND DATE '2025-09-07';
```

---

6) Quality-of-life features to add to the DSL

- **`defaults`**: set default values (e.g., **`month = today().startOfMonth()`**).
- **`constraints`**: cross-field rules (e.g., **`total == forced + scheduled`** hard check).
- **`attachments`**: photo, PDF, or meter snapshot.
- **`signoff`**: capture role + signature + timestamp per shift.
- **`print`**: page size, margins, header/footer (so you can render to a paper-like PDF).
- **`permissions`**: who can edit which section/columns (A/B/C shift).
- **`versioning`**: **`version: "1.1"`** → old instances still render with original DSL.

---

7) How to implement quickly

- **Parser**: none needed—YAML/JSON → plain POCO/DTO (C#) or TypeScript interface.
- **Renderer**: one component per widget type (**`Field`**, **`Group`**, **`Table`**, **`Grid`**, **`Checklist`**).
- **Evaluator**: tiny expression engine for **`formula`**/**`aggregates`**. (NCalc for .NET or a minimal JS eval with a safe context.)

- **DB**: start JSONB-first, then add codegen for normalized tables where needed.

If you want, I can:

- convert one full page (e.g., **Inspection & Maintenance of Power Transformer**) into the DSL completely,
- and sketch the **Postgres `CREATE TABLE`** that the DSL would generate for its table widgets.

Say the word and I'll produce the ready-to-use YAML + SQL.

> Lets add some rule - A grid can have finite rows, or inifinite rows
> Lets add the concept of Page in a Form

Perfect. Let's extend the DSL with **rules for grids** and **pages inside a form**.

---

1. Adding `grid.row_mode` (finite vs infinite)

```yaml
grid:
  rows:
    mode: "finite" | "infinite"
    generator?: { ... }        # if finite
    min?: int                  # if infinite, min rows to show initially
    max?: int                  # if finite, cap rows at max
```

- **Finite grid** → like your Monthly Shift Roster (fixed names × days).
- **Infinite grid** → like a duty log where operators can add as many rows as needed.
- UI difference: finite = pre-generated, no "add row" button; infinite = editable with "+ add row."

---

2. Adding the concept of Pages

Some of your PGCB forms span multiple physical sheets (e.g., Transformer Log has Page 1 & 2). We can define `pages[]` under `form`, each page with its own sections.

```yaml
form:
  id: transformer-log-sheet
  title: "Grid Sub-Station Transformer Log Sheet"
  version: "1.0"

  pages:
    - id: page-1
      title: "Transformer Readings"
      sections:
        - id: tr-a
          title: "Transformer TR #A"
          widgets:
            - type: table
```

```
              id: tr-a-table
              table:
                row_generators:
                  - { type: "times", start: "07:00", end: "22:00", step_minutes: 60 }
                columns:
                  - { name: time, type: time, readonly: true }
                  - { name: kv_bus_voltage, type: decimal }
                  - { name: pf_lv, type: decimal }

    - id: page-2
      title: "Shift Signatures & Counters"
      sections:
        - id: counters
          title: "OLTC and Meter Readings"
          widgets:
            - type: group
              id: oltc
              fields:
                - { name: oltc_reading, label: "OLTC Counter @12:00", type: decimal }
                - { name: energy_reading, label: "Energy Meter @12:00", type: decimal }

        - id: signatures
          title: "Shift Signatures"
          widgets:
            - type: field
              id: sig-a
              field: { name: sig_a_shift, label: "A shift", type: signature }
            - type: field
              id: sig-b
              field: { name: sig_b_shift, label: "B shift", type: signature }
```

---

3. Updated DSL Rules

✅ Grid row rules

- **`mode: finite`** = all rows defined up front (via **`generator`** or explicit list).
- **`mode: infinite`** = user may add arbitrary rows.

✅ Pages

- **`form.pages[]`** is now allowed.
- Each page has its own **`sections[]`**.
- Good for:
  - Very large forms split across sheets,
  - Reports that print page numbers automatically,
  - Multi-step UI wizards.

---

4. Example: Monthly Shift Roster with finite grid

yaml

```yaml
form:
  id: monthly-shift-duty-roster
  title: "Monthly Shift Duty Roster"
  version: "1.0"

  pages:
    - id: roster
      title: "Shift Roster"
      sections:
        - id: duty-roster
          title: "Roster"
          widgets:
            - type: grid
              id: shift-grid
              grid:
                rows:
                  mode: finite
                  generator: { type: "names" }
                columns:
                  generator: { type: "days-of-month" }
                cell:
                  type: enum
                  enum: ["A","B","C","G","F","Ad",""]
                  help: "A=06–14, B=14–22, C=22–06, G=Govt Holiday, F=Weekly, Ad=Additional"
```

---

5. Example: Substation Interruption Table with infinite rows

```yaml
form:
  id: substation-performance
  title: "Sub-Station Performance Report"
  version: "1.1"

  pages:
    - id: interruptions
      title: "Interruptions Log"
      sections:
        - id: substation-interruptions
          title: "Interruptions"
          widgets:
            - type: table
              id: interruption-table
              table:
                row_mode: infinite
                min: 1
                columns:
                  - { name: sl_no, type: integer }
                  - { name: capacity_mva, type: decimal }
                  - { name: forced, type: integer }
                  - { name: scheduled, type: integer }
                  - { name: total, type: integer, formula: "forced + scheduled" }
```

👉 With this, your DSL supports:

- **Simple fields**
- **Tables**
- **Grids (finite or infinite)**
- **Pages**

Do you want me to **draft a formal schema (JSON Schema or C# DTOs)** for this DSL so you can start coding a parser/renderer immediately?

Since we will run reports. I need store the data in table form. Lets discuss ideas

Awesome—let's lock down a storage approach that makes **reporting fast** without sacrificing flexibility.

I'll propose a **two-layer design**:

1. a small, stable **canonical schema** that stores every submission;
2. **auto-generated reporting tables** (one per table/grid widget) that are easy to query and index.

Layer 1 — Canonical (always-on) storage

These tables never change, even if forms evolve. They let you re-render any past submission, audit, and regenerate reporting tables.

Tables

- `form_definitions(form_id, version, dsl_jsonb, created_at, is_active)`
- `form_instances(instance_id, form_id, version, submitted_at, submitted_by, header_ctx jsonb, raw_data jsonb, checksum)`
    - `header_ctx` holds "header" fields (substation, month, etc.) for fast joins.
    - `raw_data` is the whole filled form (so nothing is ever lost).

  You don't report directly from these; you **generate** the reporting layer from the DSL and these snapshots.

Layer 2 — Reporting (auto-generated from the DSL)

For each `table` or `grid` widget, generate a **physical table** with flat columns. Give each a **foreign key** to `form_instances`.

Naming convention

```php-template
<snake_form_id>__<snake_widget_id>
```

Examples

- `substation_performance__substation_perf`
- `monthly_shift_duty_roster__roster_grid`

Core columns added to every reporting table

- `instance_id uuid not null references form_instances(instance_id)`

- `row_id bigint generated always as identity` (or natural row key if finite grid)
- `page_id text` (so you can filter Page 1 vs Page 2)
- `section_id text`
- `widget_id text`
- `recorded_at timestamptz default now()`
- `valid_from timestamptz default now()` / `valid_to timestamptz` (optional SCD2 for re-ingestion)
- `trace jsonb` (optional: store row-level extras, validation messages, source device, geo)

Column typing rules (DSL → SQL)

- `string/text` → `text`
- `integer` → `integer`
- `decimal` → `numeric(18,6)` (tune precision/scale per field's `unit`)
- `date` → `date`, `time` → `time`, `datetime` → `timestamptz`
- `bool` → `boolean`
- `enum` → `text` + `check (value in (...))` **or** `enum type` (Postgres native)
- `signature/attachment` → `text` (URL/blob key) + optionally `bytea` if storing inline
- Computed `formula` columns can be:
  - **(a)** materialized before insert, or
  - **(b)** a generated column: `generated always as (...) stored`

Keys & indexes

- `primary key (instance_id, row_id)` (or `(instance_id, natural_row_key...)`)
- If you frequently filter by **date/month/substation**, add columns for them and index:
  - e.g., `month date`, `substation text` extracted from `header_ctx`
- Common indexes:
  - `(month)`, `(substation)`, `(month, substation)`, `(employee_name, day)` for rosters.

Concrete examples

A) Sub-Station Performance (table widget)

DSL columns (simplified)

`sl_no, capacity_mva, forced, scheduled, total (formula), upto_30_min, upto_1_hr, more_than_1_hr, energy_mwh, remarks`

DDL

```sql
sql

create table if not exists substation_performance__substation_perf (
  instance_id uuid not null references form_instances(instance_id),
  row_id bigserial primary key,
  page_id text not null default 'interruptions',
  section_id text not null default 'a-substation',
  widget_id text not null default 'substation-perf',

  -- report filters copied from header_ctx of the instance
  substation text,
  month date,

  sl_no integer,
  capacity_mva numeric(18,3),
  forced integer check (forced >= 0),
  scheduled integer check (scheduled >= 0),
  total integer generated always as (coalesce(forced,0) + coalesce(scheduled,0)) stored,
  upto_30_min integer check (upto_30_min >= 0),
  upto_1_hr integer check (upto_1_hr >= 0),
  more_than_1_hr integer check (more_than_1_hr >= 0),
  energy_mwh numeric(18,6) check (energy_mwh >= 0),
  remarks text,

  recorded_at timestamptz default now()
);

create index if not exists ix_subperf_month on substation_performance__substation_perf(month);
create index if not exists ix_subperf_substation_month on
substation_performance__substation_perf(substation, month);
```

Ingest rule

When saving a submission:

- Insert into **`form_instances`**.
- For each row of the widget, insert into **`substation_performance__substation_perf`**, copying **`substation`** & **`month`** from **`header_ctx`**.

Typical report

```sql
sql

-- monthly energy interruption by substation
select substation,
       month,
       sum(energy_mwh) total_mkwh,
       sum(total) total_interruptions
from substation_performance__substation_perf
group by 1,2
order by 2,1;
```

B) Monthly Shift Duty Roster (finite grid)

Grid = **names × days** with enum cells `['A','B','C','G','F','Ad','']`.

Schema choices

You have two good options:

Option 1: Tall table (1 row per person per day)

```sql
create table if not exists monthly_shift_duty_roster__roster_grid (
  instance_id uuid not null references form_instances(instance_id),
  row_id bigserial primary key,
  page_id text not null default 'roster',
  section_id text not null default 'duty-roster',
  widget_id text not null default 'shift-grid',

  sub_station text,
  month date,

  person_name text not null,
  day date not null,
  shift_code text not null check (shift_code in ('A','B','C','G','F','Ad','')),

  recorded_at timestamptz default now()
);

create index if not exists ix_roster_substation_month on
monthly_shift_duty_roster__roster_grid(sub_station, month);
create index if not exists ix_roster_person_day on
monthly_shift_duty_roster__roster_grid(person_name, day);
```

Pros: simplest to query (“who worked C last Friday?”).

Cons: more rows (31 × people).

Option 2: Wide table (1 row per person, 31 columns)

Good for printing; less ideal for analytics. I strongly recommend **Option 1** for reports.

Sample report

```sql
-- coverage by shift per day
select day,
       count(*) filter (where shift_code='A') as a_shift,
       count(*) filter (where shift_code='B') as b_shift,
       count(*) filter (where shift_code='C') as c_shift
from monthly_shift_duty_roster__roster_grid
where sub_station = 'Kallyanpur' and month = date '2025-09-01'
group by day
order by day;
```

C) Transformer Log Sheet (hourly series)

Tall time-series table (recommended)

```sql
create table if not exists transformer_log_sheet__tr_a_table (
  instance_id uuid not null references form_instances(instance_id),
  row_id bigserial primary key,
  page_id text default 'page-1',
  section_id text default 'tr-a',
  widget_id text default 'tr-a-table',

  date date not null,               -- from header
  transformer_id text,              -- if present in header
  time time not null,               -- from row generator

  kv_bus_voltage numeric(10,3),
  pf_lv numeric(6,3) check (pf_lv between -1 and 1),
  mw_lv numeric(12,3),
  mvar_lv numeric(12,3),
  amp_lv_r numeric(10,3),
  amp_lv_y numeric(10,3),
  amp_lv_b numeric(10,3),
  winding_oil_temp numeric(6,2),

  recorded_at timestamptz default now()
);

create index if not exists ix_tr_a_date_time on transformer_log_sheet__tr_a_table(date, time);
```

Report example

```
sql

-- daily max oil temp per transformer
select date, transformer_id, max(winding_oil_temp) as max_temp
from transformer_log_sheet__tr_a_table
group by 1,2
order by 1,2;
```

How to keep schema in sync (codegen + migrations)

1. **Read DSL** at publish time.
2. For each `table/grid` widget, **generate**:
    - `CREATE TABLE IF NOT EXISTS ...` with columns & checks
    - `CREATE INDEX ...` from hints in DSL (`report_keys`, `index: true`)
3. Keep a **schema hash** per widget in `form_definitions`.
4. If DSL changes:
    - Generate `ALTER TABLE` migration for adds/renames (use `old_name` mapping in DSL to preserve data).
    - Bump `form.version`.
    - New submissions store `version` in `form_instances`.

    Old instances still map to old tables; or you can keep one table and only ever **add** columns (no breaking changes).

Ingestion pipeline (algorithm)

- Begin transaction
- Insert `form_instances`
- For each page/section/widget:
    - If `type=table | grid`:
        - Map each row → INSERT into generated table
            - Copy common filters (`month`, `substation`, `date`, etc.) from `header_ctx`

- Evaluate `formula` columns or let generated columns compute them
- Commit

Add **server-side validation** before insert:

- enforce `required`, `min/max`, enums, and cross-field constraints (`total == forced + scheduled`).

Extras you'll want soon

- **Materialized views** for heavy dashboards (refresh hourly).
- **Surrogate dimensions** for common entities (`dim_substation`, `dim_employee`) with FK instead of text.
- **Row provenance**: device, user, IP, geo, photo attachments (helps audits).
- **Soft deletes & re-ingest**: mark old rows `valid_to = now()` on corrections.
- **Row hashing** to detect duplicate uploads when offline clients resync.

Quick win to start

- Implement the **tall tables** for:
  1. `substation_performance__substation_perf`,
  2. `monthly_shift_duty_roster__roster_grid`.
- Build 3–4 **saved SQL views** for your most common KPIs.
- Wire the **DSL→SQL codegen** for those two and iterate.

If you want, tell me which DB you'll use (I assume **PostgreSQL**). I can output:

- JSON Schema for the DSL,
- a **C# codegen** snippet that takes DSL → `CREATE TABLE` + `INSERT`, and
- ready-to-run `CREATE VIEW` statements for your first reports.

> Lets define the dsl first so that we can start somewhere. Can you write a DSL spec for the yaml

Awesome—here's a clean, "ready-to-implement" **YAML DSL spec v0.1** for your forms. It's concise but complete, and includes a JSON Schema you can use to validate YAML files after parsing.

DSL: Forms YAML (v0.1)

Top-level

```yaml
form:
  id: <slug>                      # required. unique, kebab/snake-case
  title: <string>                 # required
  version: <semver|string>        # required, e.g., "1.0"
  locale?: [<bcp47>, ...]         # optional, e.g., ["en", "bn"]
  labels?: { <lang>: <string> }   # optional i18n display title

  meta?:                          # optional header data printed on pages/reports
    organization?: <string>
    document_no?: <string>
    effective_date?: <date>
    revision_no?: <string>
    reference?: <string>
    tags?: [<string>, ...]

  options?:                       # renderer/runtime hints (non-semantic)
    print?: { page_size?: A4|Letter, margins_mm?: {top:10,left:10,right:10,bottom:10} }
    permissions?: { roles?: [<string>], visibility?: "public"|"private" }
    storage?: { mode?: "jsonb"|"normalized" }   # preferred storage mode

  pages:                          # required (>=1)
    - (Page)
```

Page

```yaml
id: <slug>                        # required
title: <string>                   # required
labels?: { <lang>: <string> }     # optional i18n
sections:                         # required (>=1)
  - (Section)
```

Section

```yaml
id: <slug>                        # required
title: <string>                   # required
labels?: { <lang>: <string> }     # optional
widgets:                          # required (>=1)
  - (Widget)
```

Widget (discriminated by `type`)

Common properties on all widgets:

```yaml
type: field | group | table | grid | checklist
id: <slug>                        # required
title?: <string>
labels?: { <lang>: <string> }
when?: <expression>               # optional: show/hide; boolean expression
help?: <string>                   # optional UI help text
```

1) `type: field` (single input)

```yaml
field:
  name: <slug>                   # required, becomes column/key
  label: <string>                # required
  type: string|text|integer|decimal|date|time|datetime|bool|enum|attachment|signature
  required?: <bool>
  readonly?: <bool>
  placeholder?: <string>
  default?: <value>
  unit?: <string>                # "MkWh", "°C", "kV", etc.
  pattern?: <regex>              # for string
  min?: <number|date|time>       # for numeric/date/time
  max?: <number|date|time>
  enum?: [<string>, ...]         # when type: enum
  format?: <string>              # e.g., "0.000", "HH:mm"
  compute?: <expression>         # optional; value computed, then locked unless override=true
  override?: <bool>              # allow manual override of compute
```

2) `type: group` (several `field`s arranged together)

```yaml
fields: [ (FieldSpec), ... ]     # 1..N FieldSpec = same shape as field.field above (without id)
layout?: { columns?: <int> }     # UI hint (e.g., 2-column)
```

3) `type: table` (row list with named columns)

```yaml
table:
  row_mode?: infinite|finite     # default: infinite
  min?: <int>                    # initial rows for infinite
  max?: <int>                    # cap for finite (optional)
  row_key?: [<column-name>, ...] # optional natural key for upserts

  columns:                       # required (>=1)
    - name: <slug>               # required
      label: <string>            # required
      type: string|text|integer|decimal|date|time|datetime|bool|enum|attachment|signature
      required?: <bool>
      readonly?: <bool>
      unit?: <string>
      pattern?: <regex>
      min?: <number|date|time>
      max?: <number|date|time>
      enum?: [<string>, ...]     # when type: enum
      default?: <value>
      formula?: <expression>     # computed per-row (read-only in UI)
      format?: <string>

  row_generators?:               # used when row_mode: finite (or to prefill infinite)
    - type: range
      name: <slug>               # will create a generated, readonly column if not existing
      from: <int>
      to: <int>
      step?: <int>               # default 1
    - type: times
      name: <slug>
      start: "HH:mm"
      end: "HH:mm"
      step_minutes: <int>
    - type: enum
      name: <slug>
      values: [<string>, ...]
    - type: days-of-month        # uses context field `month` (date) from header
      name: day
  aggregates?:                   # footer totals/aggregations
    - name: <slug>
      label: <string>
      expr: <aggregate-expression>  # sum(col), avg(col), min(col), max(col), count(), 
countif(expr)
      format?: <string>
```

4) `type: grid` (2-D matrix, e.g., roster)

```yaml
grid:
  rows:
    mode: finite|infinite        # required
    generator?:                  # required when mode=finite
      # one of:
      type: names                # rows editable list of names (UI-controlled)
      # OR explicit:
      values: [<string>, ...]
    min?: <int>                  # initial visible rows when infinite
    max?: <int>                  # cap when finite (optional)

  columns:
    generator:                   # required
      # one of:
      type: days-of-month        # derives from a context `month` field
      # OR explicit:
      values: [<string|date|time>, ...]
      # OR time ranges:
      type: times
      start: "HH:mm"
      end: "HH:mm"
      step_minutes: <int>

  cell:
    type: string|integer|decimal|enum|bool
    enum?: [<string>, ...]       # e.g., ["A","B","C","G","F","Ad",""]
    default?: <value>
    required?: <bool>
    help?: <string>
```

5) `type: checklist` (named items with small statuses)

```yaml
checklist:
  items:
    - key: <slug>                # required
      label: <string>            # required
      type: bool|enum
      enum?: [<string>, ...]     # e.g., ["Good","Not Good"]
      default?: <value>
      required?: <bool>
```

Expressions

- **Context**: refer to same-row columns by `name`, and header/section fields via `ctx.<field>` or `header.<name>`.
- **Row formulas**: `formula: "forced + scheduled"`.

- **Aggregates**: `expr: "sum(energy_mwh)"`, `countif(col='C')`.
- Operators: `+ - * / %`, comparisons, `and/or/not`, functions: `sum, avg, min, max, count, countif, coalesce, abs, round, date_trunc, to_number`.
- Boolean `when:` visibility uses same expression engine (evaluates against header + live values).

Conventions & IDs

- All IDs (`id`, `name`) must be `[a-z0-9_-]+`.
- IDs are **stable**; renames should use `old_name:` to map migrations if you add that later.
- `date` is ISO (`YYYY-MM-DD`), `time` is `HH:mm` or `HH:mm:ss`.

Semantics for Storage/Reporting

- Each **form instance** captures:
  - `context/header` = all `field`s from any `group` in top sections you mark as header (or simply everything outside tables/grids).
  - For each `table`/`grid` widget, emit **tall rows** with:
    - `instance_id, page_id, section_id, widget_id`
    - key columns (from `row_generators` if present)
    - data columns (the defined `columns` or `cell` values)
- Recommended: copy selected header fields (e.g., `substation`, `month`, `date`) into each emitted row for indexing.

Minimal example

```yaml
form:
  id: substation-performance
  title: "Sub-Station & Line Performance"
  version: "1.0"
  meta:
    organization: "PGCB"
    document_no: "QF-GMD-06"

  pages:
    - id: p1
      title: "Header & Sub-Station"
      sections:
        - id: hdr
          title: "Header"
          widgets:
            - type: group
              id: header-fields
              fields:
                - { name: substation, label: "Substation", type: string, required: true }
                - { name: month, label: "Month", type: date, required: true }

        - id: a-substation
          title: "A) Sub-Station Performance"
          widgets:
            - type: table
              id: substation-perf
              table:
                row_mode: infinite
                min: 1
                columns:
                  - { name: sl_no, label: "Sl", type: integer }
                  - { name: capacity_mva, label: "Capacity (MVA)", type: decimal, min: 0 }
                  - { name: forced, label: "Forced", type: integer, min: 0 }
                  - { name: scheduled, label: "Scheduled", type: integer, min: 0 }
                  - { name: total, label: "Total", type: integer, formula: "forced + scheduled"
}
                  - { name: energy_mwh, label: "Energy (MkWh)", type: decimal, min: 0 }
                aggregates:
                  - { name: sum_energy, label: "Total Energy", expr: "sum(energy_mwh)" }

    - id: p2
      title: "Roster"
      sections:
        - id: roster
          title: "Monthly Shift Duty Roster"
          widgets:
            - type: grid
              id: shift-grid
              grid:
                rows:
                  mode: finite
                  generator: { type: names }     # user adds names list
                columns:
                  generator: { type: days-of-month }
                cell:
                  type: enum
                  enum: ["A","B","C","G","F","Ad",""]
                  help: "A=06–14, B=14–22, C=22–06"
```

JSON Schema (Draft 2020-12)

Use this to validate YAML (after parsing it into JSON). It covers structure & main enums. You can extend it with custom **`pattern`** checks and stricter numeric bounds.

json

```
{
  "$schema": "https://json-schema.org/draft/2020-12/schema",
  "$id": "https://example.com/form-dsl.schema.json",
  "title": "Form DSL v0.1",
  "type": "object",
  "required": ["form"],
  "properties": {
    "form": {
      "type": "object",
      "required": ["id", "title", "version", "pages"],
      "properties": {
        "id": { "type": "string", "pattern": "^[a-z0-9_-]+$" },
        "title": { "type": "string" },
        "version": { "type": "string" },
        "locale": { "type": "array", "items": { "type": "string" } },
        "labels": { "type": "object", "additionalProperties": { "type": "string" } },
        "meta": { "type": "object", "additionalProperties": true },
        "options": { "type": "object", "additionalProperties": true },
        "pages": {
          "type": "array",
          "minItems": 1,
          "items": { "$ref": "#/$defs/page" }
        }
      }
    }
  },
  "$defs": {
    "page": {
      "type": "object",
      "required": ["id", "title", "sections"],
      "properties": {
        "id": { "type": "string", "pattern": "^[a-z0-9_-]+$" },
        "title": { "type": "string" },
        "labels": { "type": "object", "additionalProperties": { "type": "string" } },
        "sections": {
          "type": "array",
          "minItems": 1,
          "items": { "$ref": "#/$defs/section" }
        }
      }
    },
    "section": {
      "type": "object",
      "required": ["id", "title", "widgets"],
      "properties": {
        "id": { "type": "string", "pattern": "^[a-z0-9_-]+$" },
        "title": { "type": "string" },
        "labels": { "type": "object", "additionalProperties": { "type": "string" } },
        "widgets": {
          "type": "array",
          "minItems": 1,
          "items": { "$ref": "#/$defs/widget" }
        }
      }
    },
    "widget": {
      "type": "object",
      "required": ["type", "id"],
      "properties": {
        "type": { "enum": ["field", "group", "table", "grid", "checklist"] },
        "id": { "type": "string", "pattern": "^[a-z0-9_-]+$" },
        "title": { "type": "string" },
        "labels": { "type": "object", "additionalProperties": { "type": "string" } },
        "when": { "type": "string" },
        "help": { "type": "string" },

        "field": { "$ref": "#/$defs/fieldSpec" },
        "fields": {
          "type": "array",
          "items": { "$ref": "#/$defs/fieldSpec" }
        },
        "layout": {
          "type": "object",
          "properties": { "columns": { "type": "integer", "minimum": 1, "maximum": 6 } }
        },

        "table": { "$ref": "#/$defs/tableSpec" },
        "grid":  { "$ref": "#/$defs/gridSpec" },
        "checklist": { "$ref": "#/$defs/checklistSpec" }
      },
      "allOf": [
        {
          "if": { "properties": { "type": { "const": "field" } } },
          "then": { "required": ["field"] }
        },
        {
          "if": { "properties": { "type": { "const": "group" } } },
          "then": { "required": ["fields"] }
        },
        {
          "if": { "properties": { "type": { "const": "table" } } },
          "then": { "required": ["table"] }
        },
        {
          "if": { "properties": { "type": { "const": "grid" } } },
          "then": { "required": ["grid"] }
        },
        {
          "if": { "properties": { "type": { "const": "checklist" } } },
          "then": { "required": ["checklist"] }
        }
      ]
    },

    "fieldSpec": {
      "type": "object",
      "required": ["name", "label", "type"],
      "properties": {
        "name": { "type": "string", "pattern": "^[a-z0-9_-]+$" },
        "label": { "type": "string" },
        "type": { "enum":
["string","text","integer","decimal","date","time","datetime","bool","enum","attachment","signat
ure"] },
        "required": { "type": "boolean" },
        "readonly": { "type": "boolean" },
        "placeholder": { "type": "string" },
        "default": {},
        "unit": { "type": "string" },
        "pattern": { "type": "string" },
        "min": {},
        "max": {},
        "enum": { "type": "array", "items": { "type": "string" } },
        "format": { "type": "string" },
        "compute": { "type": "string" },
        "override": { "type": "boolean" }
      },
      "additionalProperties": false
    },

    "tableSpec": {
      "type": "object",
      "required": ["columns"],
      "properties": {
        "row_mode": { "enum": ["infinite","finite"] },
        "min": { "type": "integer", "minimum": 0 },
        "max": { "type": "integer", "minimum": 1 },
        "row_key": { "type": "array", "items": { "type": "string" } },
        "columns": {
          "type": "array",
          "minItems": 1,
          "items": { "$ref": "#/$defs/columnSpec" }
        },
        "row_generators": {
          "type": "array",
          "items": { "$ref": "#/$defs/rowGen" }
        },
        "aggregates": {
          "type": "array",
          "items": { "$ref": "#/$defs/aggregateSpec" }
        }
      },
      "additionalProperties": false
    },

    "columnSpec": {
      "type": "object",
      "required": ["name", "label", "type"],
      "properties": {
        "name": { "type": "string", "pattern": "^[a-z0-9_-]+$" },
        "label": { "type": "string" },
        "type": { "enum":
["string","text","integer","decimal","date","time","datetime","bool","enum","attachment","signat
ure"] },
        "required": { "type": "boolean" },
        "readonly": { "type": "boolean" },
        "unit": { "type": "string" },
        "pattern": { "type": "string" },
        "min": {},
        "max": {},
        "enum": { "type": "array", "items": { "type": "string" } },
        "default": {},
        "formula": { "type": "string" },
        "format": { "type": "string" }
      },
      "additionalProperties": false
    },

    "rowGen": {
      "type": "object",
      "oneOf": [
        { "properties": { "type": { "const": "range" }, "name": { "type": "string" }, "from": {
"type": "integer" }, "to": { "type": "integer" }, "step": { "type": "integer" } }, "required":
["type","name","from","to"] },
        { "properties": { "type": { "const": "times" }, "name": { "type": "string" }, "start": {
"type": "string" }, "end": { "type": "string" }, "step_minutes": { "type": "integer" } },
"required": ["type","name","start","end","step_minutes"] },
        { "properties": { "type": { "const": "enum" }, "name": { "type": "string" }, "values": {
"type": "array", "items": { "type": "string" } } }, "required": ["type","name","values"] },
        { "properties": { "type": { "const": "days-of-month" }, "name": { "type": "string" } },
"required": ["type","name"] }
      ],
      "additionalProperties": false
    },

    "aggregateSpec": {
      "type": "object",
      "required": ["name", "label", "expr"],
      "properties": {
        "name": { "type": "string", "pattern": "^[a-z0-9_-]+$" },
        "label": { "type": "string" },
        "expr": { "type": "string" },
        "format": { "type": "string" }
      },
      "additionalProperties": false
    },

    "gridSpec": {
      "type": "object",
      "required": ["rows","columns","cell"],
      "properties": {
        "rows": {
          "type": "object",
          "properties": {
            "mode": { "enum": ["finite","infinite"] },
            "generator": { "$ref": "#/$defs/gridRowGen" },
            "min": { "type": "integer", "minimum": 0 },
            "max": { "type": "integer", "minimum": 1 }
          },
          "required": ["mode"],
          "additionalProperties": false
        },
        "columns": {
          "type": "object",
          "properties": {
            "generator": { "$ref": "#/$defs/gridColGen" }
          },
          "required": ["generator"],
          "additionalProperties": false
        },
        "cell": {
          "type": "object",
          "properties": {
            "type": { "enum": ["string","integer","decimal","enum","bool"] },
            "enum": { "type": "array", "items": { "type": "string" } },
            "default": {},
            "required": { "type": "boolean" },
            "help": { "type": "string" }
          },
          "required": ["type"],
          "additionalProperties": false
        }
      },
      "additionalProperties": false
    },

    "gridRowGen": {
      "oneOf": [
        { "properties": { "type": { "const": "names" } }, "required": ["type"] },
        { "properties": { "values": { "type": "array", "items": { "type": "string" } } },
"required": ["values"] }
```

```
      ]
    },

    "gridColGen": {
      "oneOf": [
        { "properties": { "type": { "const": "days-of-month" } }, "required": ["type"] },
        { "properties": { "type": { "const": "times" }, "start": { "type": "string" }, "end": {
"type": "string" }, "step_minutes": { "type": "integer" } }, "required":
["type","start","end","step_minutes"] },
        { "properties": { "values": { "type": "array", "items": {} } }, "required": ["values"] }
      ]
    },

    "checklistSpec": {
      "type": "object",
      "required": ["items"],
      "properties": {
        "items": {
          "type": "array",
          "items": {
            "type": "object",
            "required": ["key","label","type"],
            "properties": {
              "key": { "type": "string", "pattern": "^[a-z0-9_-]+$" },
              "label": { "type": "string" },
              "type": { "enum": ["bool","enum"] },
              "enum": { "type": "array", "items": { "type": "string" } },
              "default": {},
              "required": { "type": "boolean" }
            },
            "additionalProperties": false
          }
        }
      },
      "additionalProperties": false
    }
  }
}
```

If you want, I can turn one of your actual forms into a **fully valid YAML file** that passes this schema, and then sketch the **auto-generated SQL DDL** for its reporting tables.

We need to able to derive the table schema from yaml. Cna it be made generic

Yes — the DSL we just drafted can absolutely be made **generic enough to derive SQL table schemas** directly from YAML. The idea is:

- Each **form** → metadata only.
- Each **table/grid widget** → one relational table.
- Each **field/column** inside that widget → one SQL column, with type mapping rules.

---

1. DSL-to-SQL Mapping Rules

Base Columns (always present in reporting tables)

```sql
sql

instance_id uuid not null references form_instances(instance_id),
page_id text not null,
section_id text not null,
widget_id text not null,
row_id bigserial primary key,    -- unique row key
recorded_at timestamptz default now()
```

DSL Types → SQL Types

DSL type	SQL column type	Notes
string	text	short labels, names
text	text	long notes
integer	integer	
decimal	numeric(18,6)	configurable precision
date	date	
time	time without time zone	
datetime	timestamptz	
bool	boolean	
enum	text + check constraint	Or native Postgres ENUM if fixed
attachment	text	URI / file key
signature	text	URI / blob key

Formula Columns

- If `formula` is defined → either:
  - **Generated column** in SQL:

    ```sql
    sql

    total integer generated always as (forced + scheduled) stored
    ```

  - Or materialize before insert (depends on engine support).

Grid Widgets

- **finite** grid: one row per fixed row × column (good for rosters).
- **infinite** grid: one row per user-added row × column.

---

2. Example Conversion

DSL snippet

```yaml
yaml

widgets:
  - type: table
```

```
    id: substation-perf
    table:
      row_mode: infinite
      columns:
        - { name: sl_no, type: integer }
        - { name: capacity_mva, type: decimal }
        - { name: forced, type: integer }
        - { name: scheduled, type: integer }
        - { name: total, type: integer, formula: "forced + scheduled" }
        - { name: energy_mwh, type: decimal }
        - { name: remarks, type: text }
```

Derived SQL

```sql
create table substation_performance__substation_perf (
  instance_id uuid not null references form_instances(instance_id),
  page_id text not null default 'p1',
  section_id text not null default 'a-substation',
  widget_id text not null default 'substation-perf',
  row_id bigserial primary key,
  recorded_at timestamptz default now(),

  sl_no integer,
  capacity_mva numeric(18,6),
  forced integer,
  scheduled integer,
  total integer generated always as (coalesce(forced,0) + coalesce(scheduled,0)) stored,
  energy_mwh numeric(18,6),
  remarks text
);
```

---

3. Generic Algorithm (DSL → SQL)

1. Parse YAML into object.
2. Traverse:
    - For each `form.pages[].sections[].widgets[]`:
        - If `type=table` or `type=grid`: generate a reporting table.
3. Table name = `form_id__widget_id`.
4. Add base metadata columns.
5. Add each DSL column/field with type mapping.
6. If `formula`: generate computed column.
7. If `enum`: add `check (col in (...))`.
8. Add indexes from DSL hints (e.g., mark header fields like `month`, `substation` for indexing).

---

4. Generic Output Shapes

- **Tables** → `tall format` for analytics.
- **Grids** (finite roster):

```sql
(instance_id, row_id, person_name, day, shift_code)
```

- **Tables** (interruptions list):

```sql
(instance_id, row_id, sl_no, forced, scheduled, total, remarks)
```

---

5. Future-Proofing

We can add to DSL:

```yaml
storage:
  indexes:
    - [month]
    - [substation, month]
  copy_header: [substation, month]   # replicate header fields into row table
```

That way the **YAML drives both the form rendering and the DB schema**.

👉 Question for you: do you want me to **write a code template** (say in C# or Python) that reads the YAML DSL and emits `CREATE TABLE` DDL automatically? That way you can "compile" YAML → SQL before you start storing data.

based on all discussions above - can we design spec for a visual form designer

Yes 💡—now that we have a **DSL spec** for forms and a **storage/reporting model**, the next step is to design a **visual form designer spec**.

This would be the **UI + schema of how the designer interacts with the DSL**—so users don't hand-code YAML/JSON, but drag-drop and configure widgets.

🎨 Visual Form Designer — Spec

Think of three layers:

1. **Canvas model** — what the designer UI manipulates (pages, sections, widgets).
2. **Inspector panel model** — property editing for each widget.
3. **Export/Import** — load DSL YAML, edit visually, export DSL YAML.

---

1. Canvas Model

- **Form (root)**
    - Pages
        - Sections
            - Widgets (Field, Group, Table, Grid, Checklist)

Operations supported on canvas:

- Add/remove/reorder pages
- Add/remove/reorder sections
- Add/remove/reorder widgets inside sections
- Drag-drop placement
- Preview form live

---

2. Widget Palette

The palette contains all widget types. Each can be dragged into a section:

- **Field**
- **Group** (multi-field inline)
- **Table** (tabular rows/columns)
- **Grid** (matrix, e.g. duty roster)
- **Checklist**

Each widget has a **default config** on drop.

---

3. Inspector Panel (Widget Properties)

When a widget is selected, show editable properties (mirrors YAML DSL).

Common properties (all widgets)

- ID (`slug`)
- Title / Label (multi-language option)
- Help text
- Visibility rule (`when` expression)

Field properties

- `name`
- `type` (dropdown: string, integer, decimal, date, time, bool, enum, signature, attachment, text)
- Required? (checkbox)

- Readonly? (checkbox)
- Default value
- Placeholder
- Min/Max (for numeric/date/time)
- Pattern (regex)
- Enum values (list editor)
- Formula (expression builder with autocomplete)

Group properties

- Layout columns (1-6)
- Manage child fields inline

Table properties

- Row mode: finite / infinite
- Min / Max rows
- Columns: add/remove/rename columns
  - Column name
  - Label
  - Type
  - Required, Min, Max
  - Enum values
  - Formula
- Row generator wizard
  - Range (from–to)
  - Times (start, end, step)
  - Enum (list of row labels)
  - Days of month (linked to context)
- Aggregates (add/remove)
  - Expression editor
  - Label

Grid properties

- Row mode: finite / infinite
- Row generator (names / values)
- Columns generator (days of month, times, or explicit list)
- Cell config

    - Type
    - Enum values
    - Default
    - Required

Checklist properties

- Manage items (key, label, type, enum values, default)

---

4. Header & Meta Properties (Form-level)

- Form ID
- Title
- Version
- Organization
- Document number
- Effective date
- Revision no.
- Tags
- Storage mode (jsonb / normalized)
- Permissions (role-based)

---

5. Expression Builder (for formulas & visibility)

Instead of raw text formulas:

- Auto-suggest fields from the same scope
- Function library (sum, avg, min, max, round)
- Test expression with sample data

---

6. Pages & Navigation

- Multi-page forms represented as tabs in the designer.
- Add page → give ID & title.
- Sections inside page are collapsible blocks.
- Widgets are inside sections.

---

7. Database Mapping (Storage Hints)

Designer lets user mark:

- **Copy header fields into rows** (checkbox per field → "Promote for reporting")
- **Index hints** (checkbox → "Index this column in DB")

These hints become part of DSL:

```yaml
storage:
  copy_header: [substation, month]
  indexes:
    - [month]
    - [substation, month]
```

---

8. Output/Import

- **Export DSL**: full YAML spec
- **Import DSL**: load YAML → reconstruct canvas
- **Preview DB schema**: show generated SQL DDL for reporting tables

---

9. Example Designer Flow

1. User opens Designer → clicks “New Form”.
2. Enters metadata: Form ID, Title, Version.
3. Adds Page 1 → adds Section “Header”.
4. Drops a **Group** → adds fields `substation`, `month`.
5. Adds Section “Substation Performance”.
6. Drops a **Table** → adds columns (`forced`, `scheduled`, formula `total`).
7. In Inspector → marks `month` as “Promote for reporting”.
8. Designer auto-previews SQL table schema.
9. User exports YAML DSL.

---

10. Possible Implementation Approaches

- **Frontend Frameworks**: React (drag-drop libs like `react-dnd` or `dnd-kit`).
- **Form Rendering**: Use same DSL to preview live form (single renderer).
- **Storage**: Postgres JSONB + generated SQL.
- **DSL Schema Validation**: AJV (JS) or Newtonsoft.Json Schema (C#).

✅ With this spec, the visual designer is essentially a **GUI for YAML DSL editing**, with extras like formula builders, DB schema previews, and drag-drop sections.

Would you like me to **sketch a UI wireframe** (boxes showing Form → Page → Section → Widget → Inspector) so you can visualize how the designer might look?