



## i want to do something like this:

make a table for each of 6 inputs: stone, metal etc. the values in those tables will be shown as input in html page. if those tables are updates, so will the input values. then there will be another table that stores the skus. the old and newly generated skus would be stored in this. if any sku is already present, an error that it's already available should be shown to user on encoder page. also, there should be an extra column, in case if its stone is gemstone and carat has been added.

now, for the decoder, sku table would be used. the user will enter the sku and decoder page will show applicable fields like stone type, metal etc based on sku number. if the sku doesnt exist, show an error that says it doesnt exist.

suggest how to create the tables, link them, just basically how to make it work

You can model this cleanly with separate lookup tables for each input, a central skus table, and foreign keys linking them. Below is a concrete structure and flow you can implement with MySQL + your existing frontend.

### 1. Database tables

#### Lookup tables (for encoder dropdowns)

One table per controlled input:

```
CREATE TABLE stones (
    id      INT AUTO_INCREMENT PRIMARY KEY,
    code    CHAR(1)      NOT NULL UNIQUE, -- J, A, G, Z
    label   VARCHAR(50)  NOT NULL,
    active  TINYINT(1)  NOT NULL DEFAULT 1
);

CREATE TABLE styles (
    id      INT AUTO_INCREMENT PRIMARY KEY,
    code    CHAR(1)      NOT NULL UNIQUE, -- E, P, N, R, B, S
    label   VARCHAR(50)  NOT NULL,
    active  TINYINT(1)  NOT NULL DEFAULT 1
);

CREATE TABLE gemstones (
    id      INT AUTO_INCREMENT PRIMARY KEY,
    code    CHAR(3)      NOT NULL UNIQUE, -- AME, RUB, SAP...
    name   VARCHAR(100) NOT NULL,
    active  TINYINT(1)  NOT NULL DEFAULT 1
);
```

```

CREATE TABLE metals (
    id          INT AUTO_INCREMENT PRIMARY KEY,
    metal_type  VARCHAR(10)  NOT NULL,          -- 10KT, 14KT, SIL...
    code        CHAR(1)      NOT NULL,          -- A, B, C...
    label       VARCHAR(100) NOT NULL,
    active      TINYINT(1)  NOT NULL DEFAULT 1,
    UNIQUE KEY uniq_metal (metal_type, code)
);

CREATE TABLE sizes (
    id          INT AUTO_INCREMENT PRIMARY KEY,
    style_code  CHAR(1)      NOT NULL,          -- N / R / B
    code        CHAR(1)      NOT NULL,          -- A, B, C...
    label       VARCHAR(50) NOT NULL,          -- "16 inches (A)"
    active      TINYINT(1)  NOT NULL DEFAULT 1,
    UNIQUE KEY uniq_size (style_code, code)
);

```

Your encoder page will load options from these tables instead of hardcoded JSON. When you update any of these tables (add gemstone, deactivate metal, change label), the UI options update automatically on next load.

## Central SKU table

This table stores all generated SKUs, both old and new, and serves the decoder:

```

CREATE TABLE skus (
    id          INT AUTO_INCREMENT PRIMARY KEY,
    sku         CHAR(13)    NOT NULL UNIQUE,   -- Full SKU string
    stone_code  CHAR(1)     NOT NULL,          -- FK → stones.code
    style_code  CHAR(1)     NOT NULL,          -- FK → styles.code
    gemstone_code CHAR(3)    NULL,              -- FK → gemstones.code
    carat_code  CHAR(3)    NULL,              -- "010", "850", "12T" etc
    metal_type  VARCHAR(10) NOT NULL,
    metal_code  CHAR(1)     NOT NULL,          -- FK → metals.(metal_type, code)
    unique_id   VARCHAR(6)  NOT NULL,
    size_code   CHAR(1)     NULL,              -- FK → sizes.code
    gem_carat_raw DECIMAL(5,2) NULL,          -- extra column if gemstone + carat prov
    created_at  TIMESTAMP   NOT NULL DEFAULT CURRENT_TIMESTAMP
);

```

You can add proper foreign keys later; initially, using the codes directly keeps it simple and matches your current logic.

## 2. How the encoder will use the database

## Step A: Load dropdown data from DB

Instead of fetching JSON files, your frontend will call endpoints like:

- GET /api/stones → SELECT code, label FROM stones WHERE active = 1 ORDER BY label
- GET /api/styles
- GET /api/gemstones?search=ame (for autocomplete)
- GET /api/metals?metal\_type=14KT
- GET /api/sizes?style=N

Your existing JS that fills <select>s / autocomplete just switches from reading JSON to calling these endpoints.

## Step B: When user clicks "Generate"

1. Frontend builds the SKU exactly as you do now from input values.
2. Before finalizing, the backend checks **uniqueness**:

```
SELECT 1 FROM skus WHERE sku = ? LIMIT 1;
```

- If a row exists → respond with an error like { ok: false, error: "SKU already exists" }
- If not → insert:

```
INSERT INTO skus (
    sku, stone_code, style_code, gemstone_code, carat_code,
    metal_type, metal_code, unique_id, size_code, gem_carat_raw
) VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?);
```

3. Backend returns { ok: true, sku: "JEA010AXXXXXA" }, and encoder shows it.

**Additional safety:** enforce uniqueness at DB level so duplicates can never slip in:

```
ALTER TABLE skus ADD CONSTRAINT uniq_sku UNIQUE (sku);
```

If you attempt to insert an existing SKU, MySQL will throw a duplicate key error; you catch that in the backend and surface a "SKU already exists" message.

## 3. How the decoder will use the database

### Step A: User enters SKU on decoder page

1. Frontend sends: GET /api/sku/:sku or POST /api/decode with { sku }.
2. Backend parses or just looks up directly:

```
SELECT *
FROM skus
WHERE sku = ?;
```

- o If no row → return { ok: false, error: "SKU does not exist" }.
- o If found → return all stored fields:

```
{
  "ok": true,
  "sku": "JEA010AXXXXXA",
  "stone_code": "J",
  "style_code": "E",
  "gemstone_code": "AME",
  "carat_code": "010",
  "metal_type": "14KT",
  "metal_code": "P",
  "unique_id": "ABC123",
  "size_code": "A",
  "gem_carat_raw": 0.50
}
```

### 3. Decoder page uses these codes to:

- o Show the **human-readable text** by cross-referencing lookup tables via additional endpoints:
  - GET /api/stone/J → "Mined (J)"
  - GET /api/gemstone/AME → "AMETHYST (AME)"
  - GET /api/metal?type=14KT&code=P → "14KY - YELLOW (P)"

Or you can join on the server and return fully resolved labels in one shot:

```
SELECT
  s.sku,
  st.label      AS stone_label,
  sty.label     AS style_label,
  g.name        AS gemstone_name,
  m.label       AS metal_label,
  sz.label      AS size_label,
  s.gem_carat_raw,
  s.carat_code,
  s.unique_id
FROM skus s
LEFT JOIN stones   st  ON st.code      = s.stone_code
LEFT JOIN styles   sty ON sty.code     = s.style_code
LEFT JOIN gemstones g   ON g.code      = s.gemstone_code
LEFT JOIN metals   m   ON m.metal_type = s.metal_type
                        AND m.code      = s.metal_code
LEFT JOIN sizes    sz  ON sz.style_code = s.style_code
                        AND sz.code     = s.size_code
WHERE s.sku = ?;
```

This way, the decoder gets everything in one response and just renders it.

## 4. Flow summary

### Encoder flow

1. Page load:
  - o Fetch lookup options from stones, styles, gemstones, metals, sizes.
2. User selects values, types optional gemstone carat.
3. Frontend builds SKU string.
4. Send to backend:
  - o Backend checks if SKU exists in skus.
  - o If exists → error "SKU already exists".
  - o If not → insert full row with all codes, plus `gem_carat_raw` if applicable.
5. Frontend shows final SKU.

### Decoder flow

1. User enters SKU, clicks Decode.
2. Backend:
  - o SELECT from skus. If none → error "SKU does not exist".
  - o If found → join with lookup tables to get labels.
3. Frontend displays:
  - o Stone Type
  - o Gemstone
  - o Gem Carat (if not null)
  - o Metal Type / Metal
  - o Style / Size
  - o Unique ID

## 5. How to link everything

- Use **codes** as the stable bridge between UI, logic, and DB:
  - o Stone: J/A/G/Z
  - o Style: E/P/N/R/B/S
  - o Gemstone: 3-character code
  - o Metal: metal\_type + code
  - o Size: style\_code + code
- Encoded SKU remains your single source of truth for the 13-character string, while `skus` table stores its **decomposed parts** for easy querying, uniqueness checking, and decoding.

If you want next, a concrete **SQL schema file** plus example **API endpoint signatures** (Node/Express style) can be laid out so you can start implementing immediately.

\*\*

1. <https://stackoverflow.com/questions/37818784/how-to-synchronize-data-between-two-tables-in-different-databases-mysql-running>
2. [https://www.reddit.com/r/softwarearchitecture/comments/z32unc/how\\_to\\_synchronize\\_2\\_mysql\\_tables\\_properly/](https://www.reddit.com/r/softwarearchitecture/comments/z32unc/how_to_synchronize_2_mysql_tables_properly/)
3. <https://www.alibabacloud.com/help/en/dts/user-guide/configure-two-way-synchronization-between-mysql-databases>
4. <https://discuss.elastic.co/t/mysql-db-sync-of-all-tables-in-db-at-once/129602>
5. <https://www.sitepoint.com/community/t/how-to-sync-two-mysql-tables-within-same-database/42129>
6. <https://dev.mysql.com/doc/workbench/en/wb-database-synchronization.html>
7. <https://www.youtube.com/watch?v=GRx9UCV1wiE>
8. <https://www.optimum7.com/blog/database-synchronization-revisioning-with-mysql.html>