🛠️ pg-schemata Model Schema Design and Best Practices

This document summarizes **everything related to designing and using Model Schemas** in `pg-schemata`, with full code examples.

---

1. **Model Schema Structure**

Approach:

- Separate **columns** and **constraints** clearly in your JavaScript model schema.

🧩 Basic Model Schema Structure

```
const userSchema = {
  schema: 'public',
  table: 'users',
  columns: [
    { name: 'id', type: 'uuid', default: 'gen_random_uuid()', notNull:
true, immutable: true },
    { name: 'tenant_id', type: 'uuid', notNull: true, immutable: true },
    { name: 'email', type: 'text', notNull: true },
    { name: 'password', type: 'text', notNull: true },
    { name: 'created_at', type: 'timestamp', default: 'now()', immutable:
true }
  ],
  constraints: {
    primaryKey: ['id'],
    unique: [
      ['tenant_id', 'email']     // Composite unique constraint
    ],
    foreignKeys: [
      {
        columns: ['tenant_id'],
        references: 'admin.tenants(id)',
        onDelete: 'CASCADE'
      }
    ],
    checks: [
      { expression: 'length(email) > 3' }
    ],
    indexes: [
      { columns: ['email'] },
      { columns: ['tenant_id', 'email'] }
    ]
  }
};
```

```
module.exports = userSchema;
```

✅ Structured, clean, ready for auto-SQL generation.

---

2. **Column Field Options**

Field	Purpose
`name`	Column name
`type`	PostgreSQL data type (`text`, `uuid`, `timestamp`, etc.)
`default`	Default value (e.g., `gen_random_uuid()`, `now()`)
`notNull`	Set `NOT NULL` constraint
`immutable`	Mark fields that cannot change after insert (e.g., `id`, `tenant_id`)

---

3. **Constraint Options**

Constraint	Purpose
`primaryKey`	Single or composite primary key
`unique`	Single or composite unique constraints
`foreignKeys`	Foreign key references to other tables
`checks`	Check constraints for custom rules
`indexes`	Extra indexes for performance

---

4. **Auto-Generating CREATE TABLE SQL**

Helper Function:

```
function createTableSQL(schema) {
  const { schema: schemaName, table, columns, constraints = {} } = schema;

  const columnDefs = columns.map(col => {
    let def = `"${col.name}" ${col.type}`;
    if (col.notNull) def += ' NOT NULL';
    if (col.default !== undefined) def += ` DEFAULT ${col.default}`;
    return def;
  });

  const tableConstraints = [];

  if (constraints.primaryKey) {
    tableConstraints.push(`PRIMARY KEY (${constraints.primaryKey.map(c =>
```

```
`"${c}"`).join(', ')})`);
  }

  if (constraints.unique) {
    for (const uniqueCols of constraints.unique) {
      tableConstraints.push(`UNIQUE (${uniqueCols.map(c =>
`"${c}"`).join(', ')})`);
    }
  }

  if (constraints.foreignKeys) {
    for (const fk of constraints.foreignKeys) {
      tableConstraints.push(
        `FOREIGN KEY (${fk.columns.map(c => `"${c}"`).join(', ')})
REFERENCES ${fk.references}` +
        (fk.onDelete ? ` ON DELETE ${fk.onDelete}` : '')
      );
    }
  }

  if (constraints.checks) {
    for (const check of constraints.checks) {
      tableConstraints.push(`CHECK (${check.expression})`);
    }
  }

  const allDefs = columnDefs.concat(tableConstraints).join(',\n  ');

  const sql = `
CREATE TABLE IF NOT EXISTS "${schemaName}"."${table}" (
  ${allDefs}
);
  `.trim();

  return sql;
}
```

---

5. **Handling Immutable Fields Automatically**

Extract Immutable Columns:

```
this.immutableColumns = schema.columns
  .filter(c => c.immutable)
  .map(c => c.name);
```

✅ Use this to exclude immutable fields from `UPDATE` queries automatically.

Example Usage:

```
const updateColumns = this.columns.filter(c =>
!this.immutableColumns.includes(c));
this.updateColumnSet = new pgp.helpers.ColumnSet(updateColumns, { table:
tableConfig });
```

---

6. **Auto-Creating Indexes**

Indexes (Optional Future Step):

```
function createIndexesSQL(schema) {
  const { schema: schemaName, table, constraints = {} } = schema;
  const indexes = constraints.indexes || [];

  return indexes.map(index => {
    const cols = index.columns.map(col => `"${col}"`).join(', ');
    return `CREATE INDEX IF NOT EXISTS ON "${schemaName}"."${table}"
(${cols});`;
  }).join('\n');
}
```

✅ Call this after table creation to set up indexes.

---

7. **Auto-Create Schema If Missing**

Creating a Schema:

```
async function createSchemaIfNotExists(db, schemaName) {
  const sql = `CREATE SCHEMA IF NOT EXISTS "${schemaName}"`;
  await db.none(sql);
}
```

✅ Useful for onboarding new tenants dynamically.

---

8. **Model CRUD Ready**

Typical Insert Using ColumnSet:

```
async create(data) {
  const query = pgp.helpers.insert(data, this.insertColumnSet) + '
RETURNING *';
  return this.db.one(query);
}
```

Typical Update Using ColumnSet (ignoring immutable columns):

```
async update(id, data) {
  const condition = pgp.as.format('WHERE id = $1', [id]);
  const query = pgp.helpers.update(data, this.updateColumnSet) + ' ' +
condition + ' RETURNING *';
  return this.db.one(query);
}
```

✅ Fast, safe inserts and updates using your model schema definition.

🎯 Final Model Schema Best Practices

Topic	Best Practice
Model Layout	Separate `columns` and `constraints`
Column Fields	Always mark `immutable` and `notNull` fields clearly
UUIDs	Use UUIDs for `id` and `tenant_id`
Primary Keys	Always define explicitly
Foreign Keys	Define references cleanly
Indexes	Add for fast lookup fields (`email`, etc.)
Table Creation	Auto-generate SQL from model
Immutable Enforcement	Block updates at code level; optional DB triggers

✅ This setup ensures **fast development**, **safe data design**, and **future-proof** SaaS readiness.