🛠️ pg-schemata Design and Best Practices

This document summarizes key architectural choices, best practices, and example code snippets for building `pg-schemata`.

---

1. **Database Connection Design**

Approach:

- Use a **single shared `pg-promise` connection pool** across all tenants.

Best Practices:

- Create one global pool.
- Manage schema switching manually per tenant.

```
const pgp = require('pg-promise')();
const db = pgp({ /* connection config */ });
```

✅ No need for multiple pools.

---

2. **Tenant Schema Management**

Approach:

- Each tenant has a separate Postgres schema.
- Models switch schemas at runtime.

Best Practices:

- Call `.setSchema(schemaName)` after login.

```
userModel.setSchema('org_abc');
const user = await userModel.findById('uuid');
```

✅ Dynamic schema switching is lightweight and safe.

---

3. **Model Schema Definitions (JavaScript Model Schema)**

Approach:

- Define models in structured JavaScript.

Best Practices:

- Split `columns` and `constraints`.

```
const userSchema = {
  schema: 'public',
  table: 'users',
  columns: [
    { name: 'id', type: 'uuid', default: 'gen_random_uuid()', notNull: true, immutable: true },
    { name: 'email', type: 'text', notNull: true }
  ],
  constraints: {
    primaryKey: ['id'],
    unique: [['email']],
    indexes: [{ columns: ['email'] }]
  }
};
```

✅ Clear, extensible, future-proof.

---

4. **UUID Usage for Primary Keys and Tenant IDs**

Approach:

- Use UUIDs for both `id` and `tenant_id`.

Best Practices:

- Auto-generate UUIDs using `gen_random_uuid()`.
- Always have `tenant_id` for tenant ownership.

```
columns: [
  { name: 'id', type: 'uuid', default: 'gen_random_uuid()', notNull: true, immutable: true },
  { name: 'tenant_id', type: 'uuid', notNull: true, immutable: true }
]
```

✅ Universally unique, safe, scalable.

---

5. **ColumnSets in BaseModel**

Approach:

- Build insert and update ColumnSets separately.

Best Practices:

- Insert all columns.

- Update only mutable columns.

```
buildColumnSets() {
  const tableConfig = { table: this.table, schema: this.schema.schema };

  this.insertColumnSet = new pgp.helpers.ColumnSet(this.columns, { table:
tableConfig });

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

✅ Safer inserts and updates, reusable.

---

6. **CRUD Read Operations in BaseModel**

Approach:

- Provide essential read methods.

Best Practices:

- Cover common patterns.

```
async findById(id) {
  return this.db.oneOrNone(
    `SELECT * FROM "${this.schema.schema}"."${this.table}" WHERE id = $1`,
    [id]
  );
}

async findAll({ limit = 50, offset = 0 } = {}) {
  return this.db.any(
    `SELECT * FROM "${this.schema.schema}"."${this.table}" ORDER BY id
LIMIT $1 OFFSET $2`,
    [limit, offset]
  );
}
```

✅ Clean, efficient querying.

---

7. **Immutable Fields**

Approach:

- Enforce immutability in JavaScript (optional: enforce in database too).

Best Practices:

- Mark `immutable: true` in model schemas.
- Exclude immutable fields in updates.

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

✅ Prevents accidental overwrites of critical fields.

---

8. **Auto-Create Schema and Tables**

Approach:

- Create tenant schemas and tables programmatically.

Best Practices:

- Auto-create schemas and then tables on signup.

```
await db.none('CREATE SCHEMA IF NOT EXISTS "org_abc"');

const sql = createTableSQL(userSchema);
await db.none(sql.replace('public', 'org_abc'));
```

✅ Smooth onboarding for new tenants.

🎯 Final Design Principles

Principle	Why
Single Connection Pool	Simplicity, scalability
Schema Switching	Flexibility across tenants
Structured Model Schema	Machine-readable, safe, extensible
UUID Everywhere	Safe, scalable ID design
ColumnSets Early	Performance and safety
Dynamic Read Methods	Cover common cases cleanly

Principle	Why
Immutable Fields Managed Properly	Prevents accidental corruption
Auto-generation of SQL	Future-proof for migrations and setup

🚀 Final Thought

✅ This setup gives you real SaaS-grade multi-tenant architecture, fast, flexible, and ready to grow.