Jeffrey Yan Follow
Software Engineer and Tech Enthusiast
Mar 3 · 5 min read

Developing multi-tenant applications on Node.js and objection.js

Most Node.js API dev tutorials give an overview of REST API basics. While these are a great guide for scaffolding out your CRUD methods, I've found there aren't many resources for more complex problems. One such problem that comes up in many applications is multi-tenancy. Multi-tenancy refers to a single application instance supporting multiple *tenants*—which includes most SaaS web applications.

In this discussion I'll be focusing on the Node.js platform, as well as RDBMS (specifically Postgres) as the database technology. The usage of Postgres opens up an option to use the *schema* concept as a tenant separation mechanism, which is referred to (but not used) later.

• • •

Patterns

Multi-tenancy implementations aren't a binary concept, but rather a continuum of solutions depending on requirements. The continuum looks something like:

- An application backend per-tenant (webserver, database)
- A single webserver served by databases per-tenant
- A single webserver with a single database, served by database specific separation mechanisms (e.g. Postgres schemas)
- A single webserver with a single database, with tenant data separated into their own tables
- A single webserver with a single database, with shared tables mixing all tenant data

Each of these solutions has trade-offs in terms of:

- Data isolation and security
- Infrastructure costs and scalability
- Development complexity
- Difficulty of onboarding new tenants
- Difficulty of modifying data structures for existing tenants

These trade-offs are referred to in this detailed stackoverflow post, which has some useful linked references.

The single table design

Unfortunately there isn't time to delve into every single implementation in this article. I'll be specifically referring to the shared-table multi-tenancy pattern, sometimes referred to as row-based multi-tenancy. This pattern is more ideal for situations where there are **many tenants each with relatively small amounts of data**. This post summarizes some trade-offs well.

When using Postgres, the option of **schemas** and **views** opens up the possibility of a hybrid design—where a schema will exist per tenant, with views providing a slice of the tenant data accordingly. Postgres views are essentially treated as stored queries, since they don't store any data. The concept is useful but it does add another layer of detail to maintain (the schemas). These **view queries** look something like:

```
CREATE VIEW "customers_view_ABCD" AS (
  SELECT *
  FROM "customers"
  WHERE "tenantId" = 'ABCD'
)
```

This query is relatively trival to generate dynamically, so doesn't seem worth the maintenance complexity.

objection.js

The reason that objection.js was chosen as the ORM of choice was due to it also exposing the powerful knex.js interface. The power of the ORM can be leveraged when doing CRUD style queries such as find or

insert. Then the knex.js query builder can be used when more power is required.

The best part about objection.js though is that these 2 styles can be used simultaneously in the same query. This leads to some potentially powerful filtering mechanics which aren't discussed here. I ended up creating a filtering convenience package to leverage some of these features (shameless plug!).

. . .

Design and Implementation

In terms of making the application *multi-tenant aware*, these concepts should exist in both the **database** as well as the **API**. Some implementations add a *tenantId* column onto every tenant related table. Others will only add the column onto tables which are adjacent to the tenant model, with related models inferring tenancy from upper level models. For the purpose of simplicity, here we'll assume that every relevant table has a *tenantId* column.

Lets say that our application has 4 models:

- Tenant
- Shop
- Product
- Country

Tenants sign up to our application to manage their shops and products. Given this, the **Tenant, Shop** and **Product** models are tenant-aware. On top of these, each **Shop** has a relation to the **Country** model which is not tenant specific (so has no *tenantId* column).

. . .

Retrieving data

Most likely the **GET /Tenants** endpoint will be protected, since you don't really want anyone outside of application admins getting these. With that in mind, our REST API is going to look something like:

```
GET /Shops
GET /Products
GET /Countries
```

Since shops and products are tenant specific, there'll need to be a mandatory filter on the *tenantId:*

```
GET /Shops?tenantId=ABCD
GET /Products?tenantId=ABCD
GET /Countries
```

That is about the minimal API surface we'll need to query our entire application. I decided on Hapi.js due to it's recent implementation of async/await, but any API framework would be similar. An API endpoint handler would then look something like:

```
// GET /Shops
const Model = require('./shop-objection-model');
async function getShops(request) {
  const { tenantId } = request.query;
  return await Model.query().where({ tenantId });
};
```

This query results in a generated SQL query which is exactly the same as the Postgres **view** described earlier—slicing the shops model by tenantId.

```
-- Create views for the ABCD tenant
CREATE VIEW "shops_view_ABCD" AS (
  SELECT *
  FROM "shops"
  WHERE "tenantId" = 'ABCD'
)
```

```
CREATE VIEW “products_view_ABCD” AS (
  SELECT *
  FROM “products”
  WHERE “tenantId” = ‘ABCD’
)
```

Instead of doing the SELECT into the view, we’re generating the query dynamically.

Complex data retrieval

It’s a very common task to ask for more related data when doing an API query. This is the foundation of Graphql, to minimize the amounts of overall queries. Including related data is often referred to as eager-loading and naming will vary depending the ORM.

The objection.js framework provides a built in eager loading function on top of queries, as long as model relationships are defined accordingly. Lets say that we want to get a bit more data, maybe shops and their associated country. An API query could look something like:

```
GET /Shops?tenantId=ABCD&eager=country
```

This would return data in the format of:

```
[{
  "id": 1,
  "tenantId": "ABCD",
  "name": "shop 1",
  "countryId": 10,
  "country": {
   "id": 10,
   "name": "New Zealand"
  }
 },
 {
  "id": 2,
  "tenantId": "ABCD",
  "name": "shop 2",
  "countryId": 20,
  "country": {
   "id": 20,
   "name": "Australia"
```

```
    }
  }
]
```

This ends up being a trivial implementation in the endpoint handler:

```
const Model = require('./models/shop');
const handler = function(request) {
  const { tenantId, eager } = request.query;
  return Model
    .query()
    .where({ tenantId })
    .eager(eager);
};
```

Since we're only *eagering* in the country model, this doesn't require any special protections. When we want to include related models which may be tenant specific then additional logic is required.

Maintaining isolation

With languages such as Graphql it can be more powerful to give the user flexibility when it comes to their queries. The ultimate form of flexibility would be to **query any model and include any arbitrarily deep relations**.

In order to do this, we need to know which models are tenant specific. This can be done with a static flag on the model definition:

```
class Shop extends Model {
  static get isTenantSpecific() {
    return true;
  }
}
```

The objection.js query builder then provides a query build hook perfect for this situation. The end result will be something like "during any query or eager query, if the model is tenant specific the slice it accordingly". The resultant handler changes slightly to become:

```
const Model = require('./models/shop');
const handler = function(request) {
  const { tenantId, eager } = request.query;
  return Model
    .query()
    .eager(eager)
    .context({
      onBuild: builder => {
        if (builder._modelClass.isTenantSpecific)
          builder.where({ tenantId });
      }
    })
};
```

Notice the top level *where({ tenantId })* on the model has been removed. This can now be handled by the context hook. This will result in eager queries which would be very similar to SELECTing into postgres VIEWs as shown earlier.

Now the full power of eager loading can be taken advantage of, while maintaining tenant isolation!

• • •

Summary

Overall a pattern to implement row-based multi-tenancy was discussed here. REST was used as a querying medium but GraphQL could equally be used with the same approach. Whether row-based multi-tenancy or some other type is preferred, the objection.js ORM provides a powerful platform for implementation.