Node Postgres Relationships

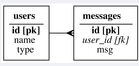
Springboard

Download Demo Code

Goals

- Work with 1:M relationships in ***pg***
- Work with M:M relationships in ***pg***
- Handle missing data by sending 404s

One to Many Relationships




```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name TEXT NOT NULL,
    type TEXT NOT NULL
);

CREATE TABLE messages (
    id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL REFERENCES users,
    msg TEXT NOT NULL
);
```

We want our API to include:

GET /users/1

Return detail of user *and* list of message:

```
{
  name: "Juanita",
  type: "admin",
  messages: [
    {id: 1, msg: 'msg #1'},
    {id: 2, msg: 'msg #2'}
  ]
}
```

GET */users/[id]* With Messages

demo/routes/users.js

```
/** Get user: {name, type, messages: [{msg, msg}]} */

router.get("/:id", async function (req, res, next) {
  try {
    const userRes = await db.query(
          `SELECT name, type FROM users WHERE id=$1`,
        [req.params.id]);

    const messagesRes = await db.query(
          `SELECT id, msg FROM messages
             WHERE user_id = $1`,
        [req.params.id]);

    const user = userRes.rows[0];
    user.messages = messagesRes.rows;
    return res.json(user);
  }

  catch (err) {
    return next(err);
  }
});
```

(results)

```
{
  name: "Juanita",
  type: "admin",
  messages: [
    {id: 1, msg: 'msg #1'},
    {id: 2, msg: 'msg #2'}
  ]
}
```

We just add property on user and populate with messages!

Note: Optimizing this code

For ease of readability, we've awaited two database queries sequentially in the above code example. We could have just as easily run these queries in parallel by wrapping them in a `Promise.all`, since the message query doesn't depend on the result of the user query.

You might also be wondering why we don't use a join, and simply make one request to the database. What would be some advantages to using this approach? What might some disadvantages be?

Many to Many Relationships


```
CREATE TABLE tags (
    code TEXT PRIMARY KEY,
    tag TEXT UNIQUE
);

CREATE TABLE messages_tags (
    message_id INTEGER NOT NULL REFERENCES messages,
    tag_code TEXT NOT NULL REFERENCES tags,
    PRIMARY KEY(message_id, tag_code)
);
```

We want our API to include:

GET /messages/1

Return info about message *and* associated tag names:

```
{
  id: 1,
  msg: "msg #1",
  tags: ["Python", "JavaScript"]
}
```

What about this query?

```
SELECT m.id, m.msg, t.tag
  FROM messages AS m
    LEFT JOIN
       messages_tags AS mt
    ON m.id = mt.message_id
    LEFT JOIN
       tags AS t
    ON mt.tag_code = t.code
  WHERE m.id = 1;
```

id	msg	tag
1	msg #1	Python
1	msg #1	JavaScript

Restructuring This Data

we get from database

```
[
  {id: 1, msg: "msg #1", tag: "Python"},
  {id: 1, msg: "msg #1", tag: "JavaScript"},
]
```

we want to return

```
{
  id: 1,
  msg: "msg #1",
  tags: ["Python", "JavaScript"]
}
```

demo/routes/messages.js

```
/** Get message: {id, msg tags: [name, name]} */

router.get("/:id", async function (req, res, next) {
  try {
    const result = await db.query(
          `SELECT m.id, m.msg, t.tag
             FROM messages AS m
               LEFT JOIN messages_tags AS mt
                 ON m.id = mt.message_id
               LEFT JOIN tags AS t ON mt.tag_code = t.code
               WHERE m.id = $1;`,
        [req.params.id]);

    let { id, msg } = result.rows[0];
    let tags = result.rows.map(r => r.tag);

    return res.json({ id, msg, tags });
  }

  catch (err) {
    return next(err);
  }
});
```

Note: Don't Forget about the debugger!

Remember, if this code starts to become too hard to track, you can use the ***debugger*** to pause code execution and see what's going on!

Note: Manipulating data

When it comes to handling these many-to-many relationships, you'll find that you often need to manipulate arrays of objects in JavaScript. There are many helper libraries with utilities that can assist with this process (such as lodash or underscore), but for now, we'll focus on writing all of the business logic ourselves.

Handling Missing Resources

We want:

PUT /messages/[id]

Given `{msg}`, updates DB & return `{id, user_id, msg}`

demo/routes/messages.js

```
/** Update message: {msg} => {id, user_id, msg} */

router.put("/:id", async function (req, res, next) {
  try {
    const result = await db.query(
          `UPDATE messages SET msg=$1 WHERE id = $2
           RETURNING id, user_id, msg`,
        [req.body.msg, req.params.id]);

    return res.json(result.rows[0]);
  }

  catch (err) {
    return next(err);
  }
});
```

Just returns ***undefined*** if not found!

(works, but requires two queries)

```
/** Update message #2: {msg} => {id, user_id, msg} */

router.put("/v2/:id", async function (req, res, next) {
  try {
    const checkRes = await db.query(
          `SELECT id FROM messages WHERE id = $1`,
        [req.params.id]);

    if (checkRes.rows.length === 0) {
      throw new ExpressError("No such message", 404);
    }

    const result = await db.query(
          `UPDATE messages SET msg=$1 WHERE id = $2,
           RETURNING id, user_id, msg`,
        [req.body.msg, req.params.id]);

    return res.json(result.rows[0]);
  }

  catch (err) {
    return next(err);
  }
});
```

(same thing, but with one query)

```
/** Update message #3: {msg} => {id, user_id, msg} */

router.put("/v3/:id", async function (req, res, next) {
  try {
    const result = await db.query(
          `UPDATE messages SET msg=$1 WHERE id = $2
           RETURNING id, user_id, msg`,
        [req.body.msg, req.params.id]);

    if (result.rows.length === 0) {
      throw new ExpressError("No such message!", 404);
    }

    return res.json(result.rows[0]);
  }

  catch (err) {
    return next(err);
  }
});
```