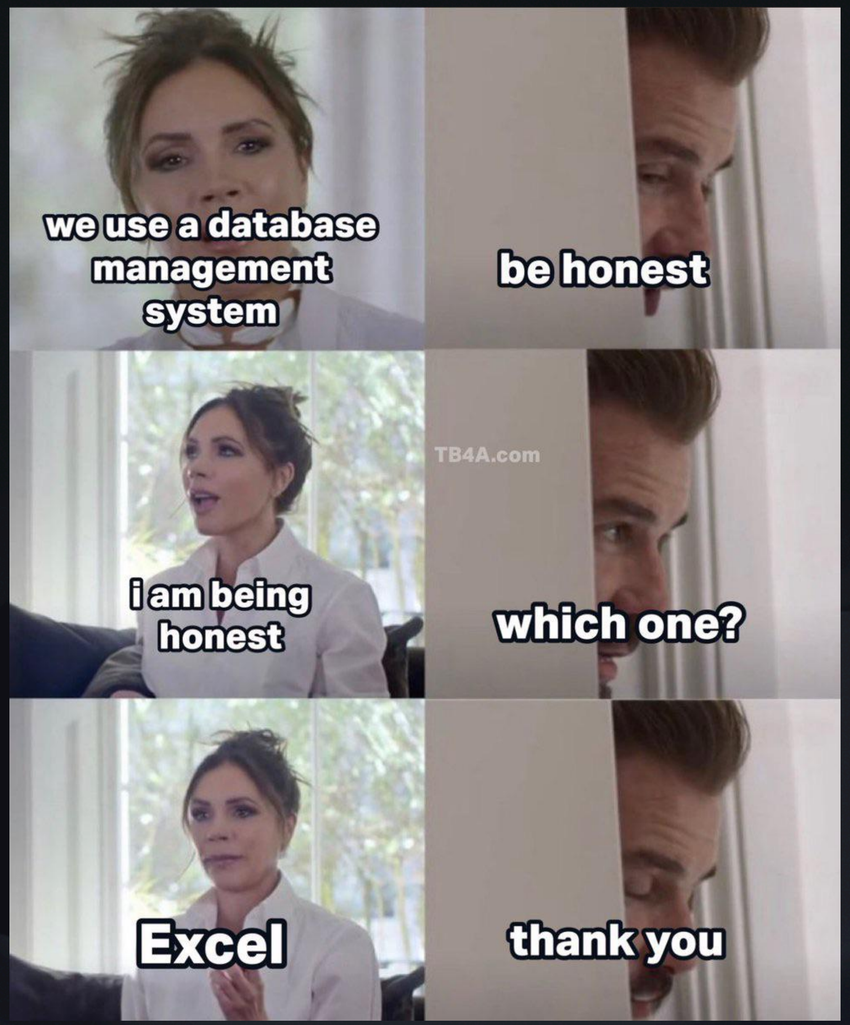
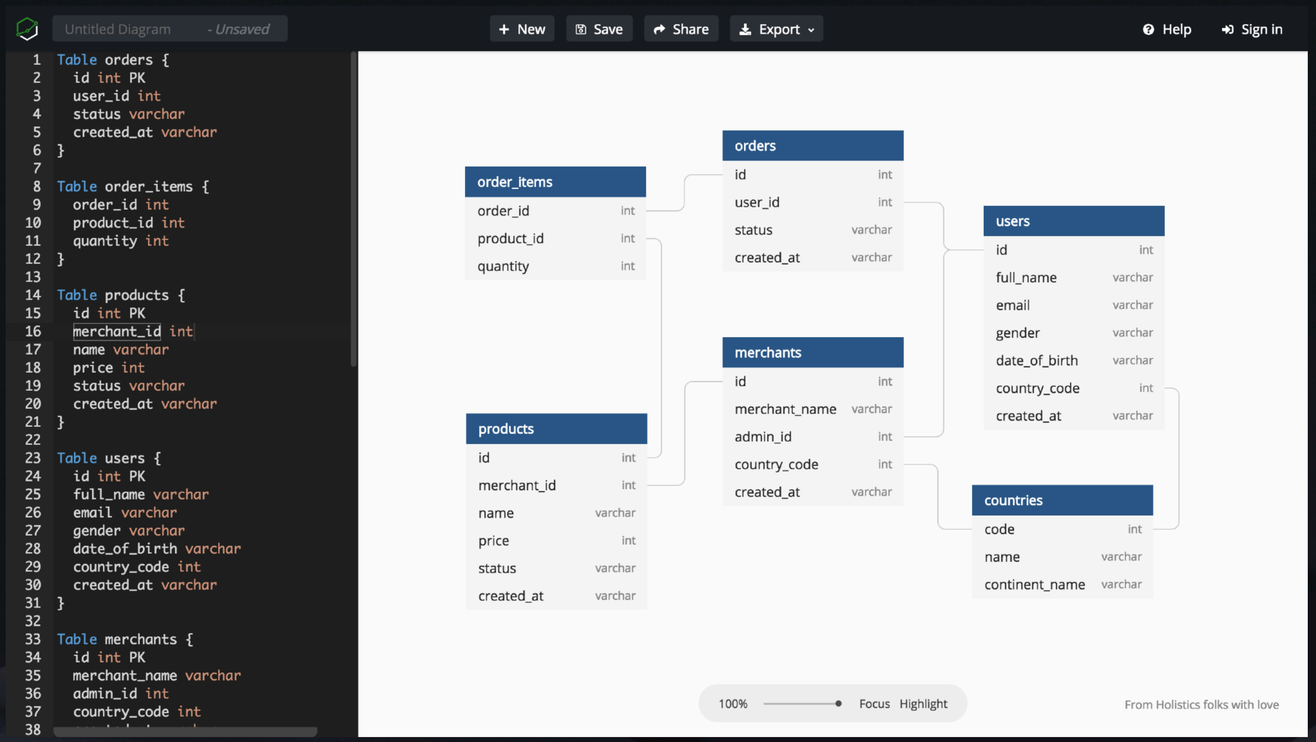
PostgreSQL & Sequelize 🗄️

Organizing & Persisting Data

Why do we need databases?



Why do we need databases?



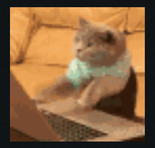
- When our program stops running, all of the variables, arrays, objects, etc are flushed from memory
- A database allows us to *persist* information
- Databases make it easier to store, search and retrieve the data we need
  - Efficiently!

SQL vs NoSQL

- SQL stands for Structured Query Language
- SQL insists on a defined schema: all data must follow all the rules in the schema
- If you want to add data that doesn’t adhere to the schema, change the schema!
- NoSQL permits flexible data, fewer restrictions on schema
- NoSQL: the developer has to take responsibility for maintaining data validity

SQL vs NoSQL

I'd like to save this data please



```
{
  "name": "Darkwing Duck",
  "jobTitle": "Vigilante",
  "pondId": 31,
  "sideKickId": 64,
}
```

I don't have a column called jobTitle

Also, there's no pond with id = 31

mongoDB

This doesn't look like the other data I have...

Whatever, I'm sure it's fine!

PostgreSQL Schema

We'll be focusing on PostgreSQL

Unlike JavaScript, PostgreSQL

Is VERY strict about types

- SQL organizes data into tables
- Each table has columns
- Each column has a type, and various properties
- Some columns reference other tables
  - "Foreign key"

```
CREATE TABLE "duck" (
    id SERIAL PRIMARY KEY,
    name character varying NOT NULL,
    username character varying NOT NULL UNIQUE,
    pond_id integer NOT NULL REFERENCES "pond"(id),
    created_at timestamp without time zone NOT NULL DEFAULT now(),
    updated_at timestamp without time zone NOT NULL DEFAULT now(),
);
```

PostgreSQL Schema

ducks table

Id	name	username	pond_id
1	DarkWing	darkwing111	2
3	Daffy	daffpunk	2
7	DarkWing	dark_wing2	1
11	Donald	don	5

ponds table

Id	name	total_area	lat_long
1	Lake Michigan	22406	[-123.4,86.9]
2	Winnecone	4	[-345.1, 12.64]
5	Green Lilly	24	[90, -12.89]

PostgreSQL Queries

```sql
-- Fetch all ducks' ids and usernames
SELECT id, username FROM "duck";
```

```sql
-- Fetch all ducks in a pond
SELECT * FROM "duck" WHERE "pond_id" = 2;
```

```sql
-- Fetch all ducks, including the pond name
SELECT
  d.id,
  d.username,
  p.name AS pond_name
FROM
  "duck" d
  JOIN "pond" p ON d.pond_id = p.id;
```

- SELECT retrieves data from the database
- We must specify which columns we want returned
- We must also specify what table we're interested in
- WHERE lets us filter the results
- JOIN lets us combine related tables

PostgreSQL Queries

```sql
-- Create a new duck
INSERT INTO "duck" (name, username, pond_id) VALUES ('Donald Duck', 'donald', 1);
```

```sql
-- Update a duck's name
UPDATE "duck" SET name = 'Daffy Duck' WHERE id = 11;
```

```sql
-- Delete a duck
DELETE FROM "duck" WHERE id = 7;
```

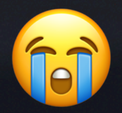
- SQL also allows us to create, update, and delete rows
- Pay VERY CLOSE ATTENTION to the WHERE clause when updating or deleting

PostgreSQL Queries

express

```
// GET all ducks
router.get("/", async (req, res) => {
  const result = await pg.query("SELECT * FROM ducks");
  res.send(result.rows);
});
```

```
// update a duck's name
router.patch("/:id", async (req, res) => {
  const { id } = req.params;
  const { name } = req.body;
  const result = await pg.query("UPDATE ducks SET name = $1 WHERE id = $2", [name, id]);
  res.send(result.rows);
});
```



PostgreSQL 🤝 Sequelize

PostgreSQL thinks in terms of

rows and relations

✨ ✨

✨ ✨

JS thinks in terms of

objects and their properties

Sequelize is an

Object-Relational Mapper (ORM)

PostgreSQL 🤝 Sequelize

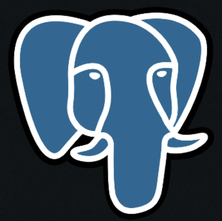
```
await pg.query("SELECT * FROM ducks")
```

```
await Duck.findAll();
```

```
const result = await pg.query(
  "INSERT INTO ducks (name, username, pond_id) VALUES ($1, $2, $3) RETURNING *",
  ["Donald Duck", "donald", 1]
);
```

```
const result = await Duck.create({
  name: "Donald Duck",
  username: "donald",
  pond_id: 1,
});
```

Sequelize: Connecting to PostgreSQL



Our Express server is a *client with respect to PostgreSQL*

```
const db = new Sequelize("postgres://localhost:5432/duck_pond");
```

```
const runApp = async () => {
  await db.sync();
  console.log("✅ Connected to the database");
  app.listen(PORT, () => {
    console.log(`🚀 Server is running on port ${PORT}`);
  });
};
runApp();
```

`db.sync()` attempts to connect to the PostgreSQL server

Sequelize: Creating Tables

```
const { DataTypes } = require("sequelize");
const db = require("./db");

const Duck = db.define("duck", {
  id: {
    type: DataTypes.INTEGER,
    autoIncrement: true,
    primaryKey: true,
  },
  name: {
    type: DataTypes.STRING,
    allowNull: false,
  },
  username: {
    type: DataTypes.STRING,
    allowNull: false,
    unique: true,
  },
  pond_id: {
    type: DataTypes.INTEGER,
    allowNull: false,
  },
});
```

- Sequelize has many different `DataTypes` – check the documentation
- A `primaryKey` should be unique and unchanging – necessary for relationships with other tables
- There are many different validations, such as allowNull, unique, isEmail, isIn, etc.

Sequelize: Associating Tables

```
const Pond = db.define("pond", {
  id: {
    type: Sequelize.INTEGER,
    autoIncrement: true,
    primaryKey: true,
  },
  // ...
});

const Duck = db.define("duck", {
  id: {
    type: Sequelize.INTEGER,
    autoIncrement: true,
    primaryKey: true,
  },
  pond_id: {
    type: Sequelize.INTEGER,
    allowNull: false,
  },
  // ...
});

Duck.belongsTo(Pond);
Pond.hasMany(Duck);
```

- Tables can be associated with each other by creating a foreign key column
  - Foreign key = Key referring to the primary key of the other table
- `belongsTo` adds a foreign key to the table on the left, referring to the table on the right: Duck gets a foreign key to Pond
- hasMany tells Sequelize that we may want to get all the Ducks associated with a pond

Sequelize: Associating Tables

There are three basic types of relationships:

One to One

```
Owner.hasOne(Cat);
Cat.belongsTo(Owner);
```

One to Many

```
Duck.belongsTo(Pond);
Pond.hasMany(Duck);
```

Many to Many

```
Student.belongsToMany(Course, { through: "Enrollment" });
Course.belongsToMany(Student, { through: "Enrollment" });
```

Many-To-Many requires that we make a whole new table, a “join table”

Sequelize now knows how to include rows from the associated table

```
const duck = await Duck.findByPk(11, { include: Pond });
console.log(duck.pond);
```

Sequelize: Class & Instance Methods

One to Many

```
Duck.belongsTo(Pond);
Pond.hasMany(Duck);
```

Class Methods

```
// find a duck
const duck = await Duck.findByPk(1);

// create a duck
await Duck.create({
  name: "Donald Duck",
  username: "donald",
  pond_id: 1,
})

// find all ducks with the names that start with "D"
const ducks = await Duck.findAll({
  where: { name: { [Op.like]: "D%" } },
});
```

Instance Methods

```
// update a duck
await duck.update({ name: "Daffy Duck" });

// destroy a duck
await duck.destroy();

// update a duck's fk
await duck.setPond(2);
```

Magic Methods

Sequelize: Tips and Tricks

- It's useful to create a seed file: populate the database with sample data
- `db.sync({ force: true })` will erase all current data and start over

```
const seed = async () => {
  await db.sync({ force: true });
  const ponds = await Pond.bulkCreate([
    {
      name: "Duck Pond",
      total_area: 100,
      lat_long: [37.7749, -122.4194],
    },
    { … },
    { … },
  ]);
  await Duck.bulkCreate([
    {
      name: "Donald Duck",
      username: "donald",
      pond_id: 1,
    },
    { … },
    { … },
  ]);
  console.log("🌱 Seeded the database");
  db.close();
};
```

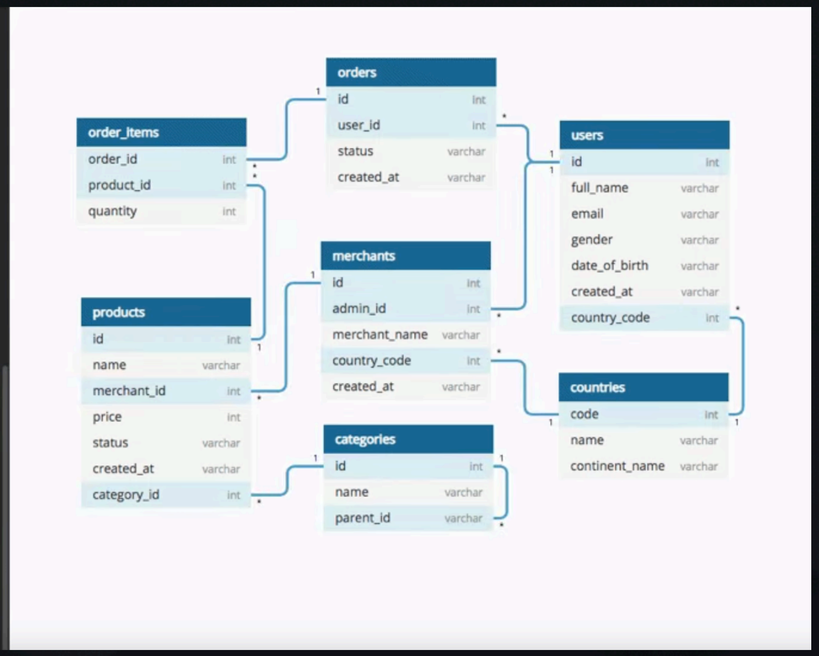
Sequelize: Tips and Tricks

- Now that we're using Sequelize within Express, we should remember:
- Always make our route handlers **`async`**, so we can use **`await`**
- Wrap our route handlers with **`try / catch`** in case an error occurs

```
router.get("/", async (req, res) => {
  try {
    const ducks = await Duck.findAll();
    res.send(ducks);
  } catch (error) {
    res.status(500).send({ error: error.message });
  }
});

router.post("/", async (req, res) => {
  try {
    const duck = await Duck.create(req.body);
    res.send(duck);
  } catch (error) {
    res.status(500).send({ error: error.message });
  }
});
```

Sequelize: Tips and Tricks

```
code int PK
name varchar
continent_name varchar


f {
orders.user_id > users.id

f {
order_items.order_id > orders.id


f {
order_items.product_id > products.id


f {
products.merchant_id > merchants.id


f {
users.country_code > countries.code


f {
merchants.admin_id > users.id

f {
merchants.country_code > countries.code


ble categories {
id int
name varchar
parent_id varchar
```



dbdiagram.io

- Designing your database schema can be challenging
- Use a diagramming tool, and talk to your teammates so that everyone is on the same page

Sequelize Practice