Databases & ORMs

Object Relational Mapper

- Acts as a “bridge” between your code and the RDBMS.
- Using ORM, data can be easily stored and retrieved from a database without writing SQL statements directly.

Sequelize

- **Sequelize is an Object-Relational Mapper (ORM)**
- **Access SQL databases from Node.js**
  - Using JS objects and methods instead of SQL statements
- Represents tables as "classes" and rows as objects (instances)

We are going to be making a simplified version of wikipedia today. We are going to build it on Node, make our routes with Express, and utilize a postgres database (i.e. using the pg module), but this time you will not be using the pg module directly meaning you will not be writing raw SQL queries nor will you be dealing with the raw returned values of those queries. How are we going to do this? Well we are going to utilize a library created that writes queries for us. It is called Sequelize. In our node application it gives to us a number of objects and methods and maps these to communicate with pg about relations. And on the way back it will parse the data for us and give us an object to work with in javascript. What we are used to. This is what I promised the other day and now you will finally get to use it.

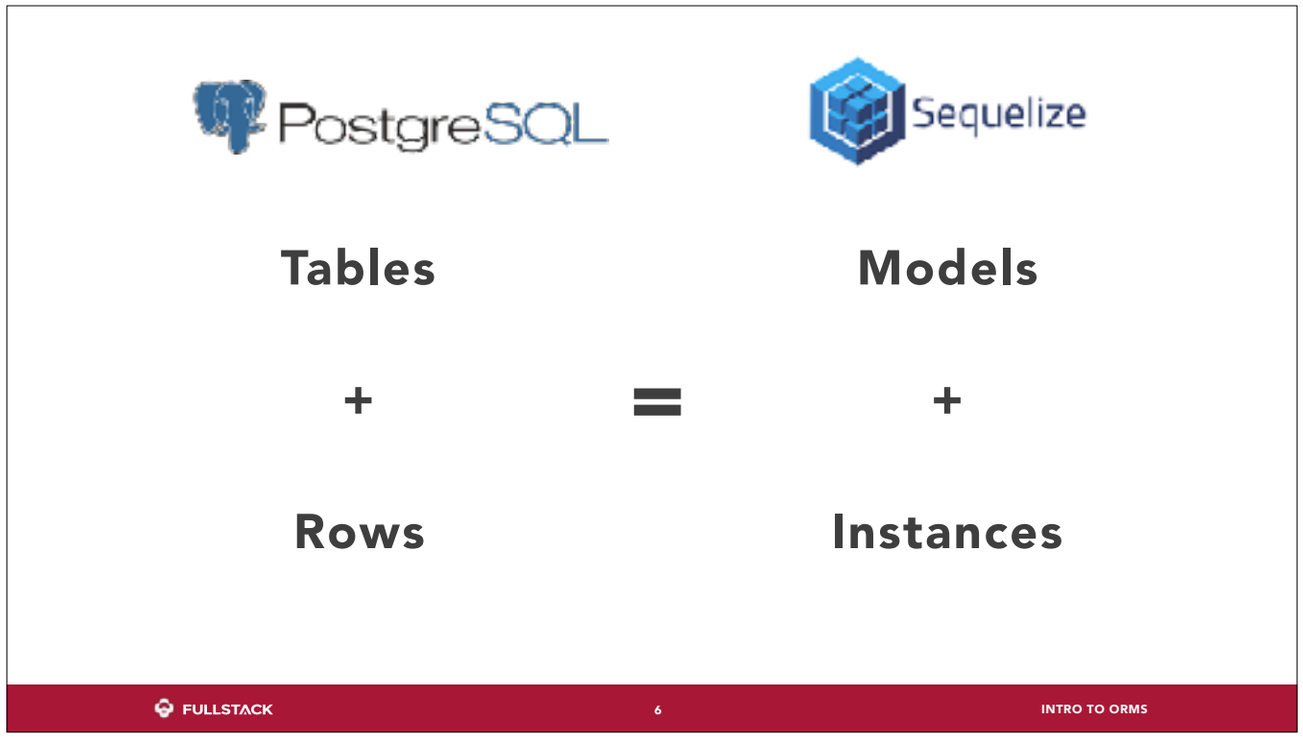
This allows for some great abstractions that save us time and heartache, but do note that there is no such thing as a perfect object relational mapper, so there maybe be some things it cannot do for us. You probably won't see this at all unless you have a capstone that delves into sequelize and databases in a deep way. It does give us 99% of everything we want to do, and it allows us to write our database interactions in a javascript way. It gives us ways to keep our database clean and very maintained.

Without ORM

```
client.query(`select * from dogs`)

client.query(`select * from cats`)

client.query(`select * from hippos`)
```

With ORM

```
Dog.findAll()

Cat.findAll()

Hippo.findAll()
```



Sequelize has 2 classes of object. Every model represents a corresponding table in your database. Sequelize also gives each model some methods and properties by default. What are examples of things we might want to do with tables? Create, delete, search them, update, etc.

If we do Users.findById() with a specific id, we will be returned an instance object. A row in the table. A single user. We can get an instance object by querying the database, but we can also create one before it ever talks to the database with build which you will see in the workshop.

Basic Workflow

Sequelize Basics: Workflow

- Instantiate Sequelize

```
const Sequelize = require('sequelize')
const db = new Sequelize('postgres://localhost/wiki')
```

Sequelize Basics: Workflow

- Instantiate Sequelize

```
const Sequelize = require('sequelize')
const db = new Sequelize('postgres://localhost/wiki')
```

- Define your Model(s)
  - Add options to Model fields (validations, default values & more)

```
const User = db.define('user', {
  name: Sequelize.STRING,
  pictureUrl: Sequelize.STRING
});
```

Sequelize Basics: Workflow

- Instantiate Sequelize

```
const Sequelize = require('sequelize')
const db = new Sequelize('postgres://localhost/wiki')
```

- Define your Model(s)
  - Add options to Model fields (validations, default values & more)

```
const User = db.define('user', {
  name: {
    type: Sequelize.STRING,
    allowNull: false
  },
  pictureUrl: Sequelize.STRING
});
```

Sequelize Basics: Workflow

- **Instantiate Sequelize**

```
const Sequelize = require('sequelize')
const db = new Sequelize('postgres://localhost/wiki')
```

- **Define your Model(s)**
  - Add options to Model fields (validations, default values & more)

```
const User = db.define('user', {
  name: {
    type: Sequelize.STRING,
    allowNull: false
  },
  pictureUrl: Sequelize.STRING
});
```

- **Connect/sync** the **Model** to an ***actual*** **table** in the **database**

```
await User.sync()
```


The

Sequelize Basics: Workflow

- **Use** the **Model** (Table) to **find/create Instances** (row)

```
const users = await User.findAll();
```

Sequelize Basics: Workflow

- Use the **Model** (Table) to find/create **Instances** (row)
- Use the **Instances** to save / update / delete

```
const person = new User({
  name: "Kate",
  pictureUrl: "http://fillmurrary.com/10/10"
});
```

```
await person.save();
```

Additional Model Options

- Sequelize models can be extended Hooks, Class & Instance Methods, Getter & Setters, Virtuals, etc.

In the workshop, you willl work with Hooks.
Hooks give us the ability to "hook" into Sequelize operations (like updating, creating, or destroying an instance) and execute arbitrary functions related to that instance.

This can be useful in several ways. For example: Before creating an instance of a User, we could hash that user's plaintext password, so that what gets saved is the hashed version

We’re not going to use the other examples right now, but just for a quick definition:
Class method — Example: findAll Kates
Instance method — format a phone number
Getter & Setters: Override what happens when someone 'gets' or 'sets' a property on an instance
Virtuals — Example: Getter for fullname (create new attribute each time I query this table)

Hooks

Hooks

- When you perform various operations in Sequelize (creating, updating, destroying, etc), various "events" occur. These are called "lifecycle events"

- Hooks are like adding an event listener to these events
  - *"Every time a journal entry is created or updated, escape any dangerous sequences that could result in an XSS attack"*
  - *"Every time a user is updated with a new password, hash it so that the plaintext password doesn't get saved in the database"*

What happens when we do this?

```
const pug = await User.create({
  name: “Cody",
  pictureUrl: "http://fillmurray.com/10/10"
});
```

beforeValidate

validation

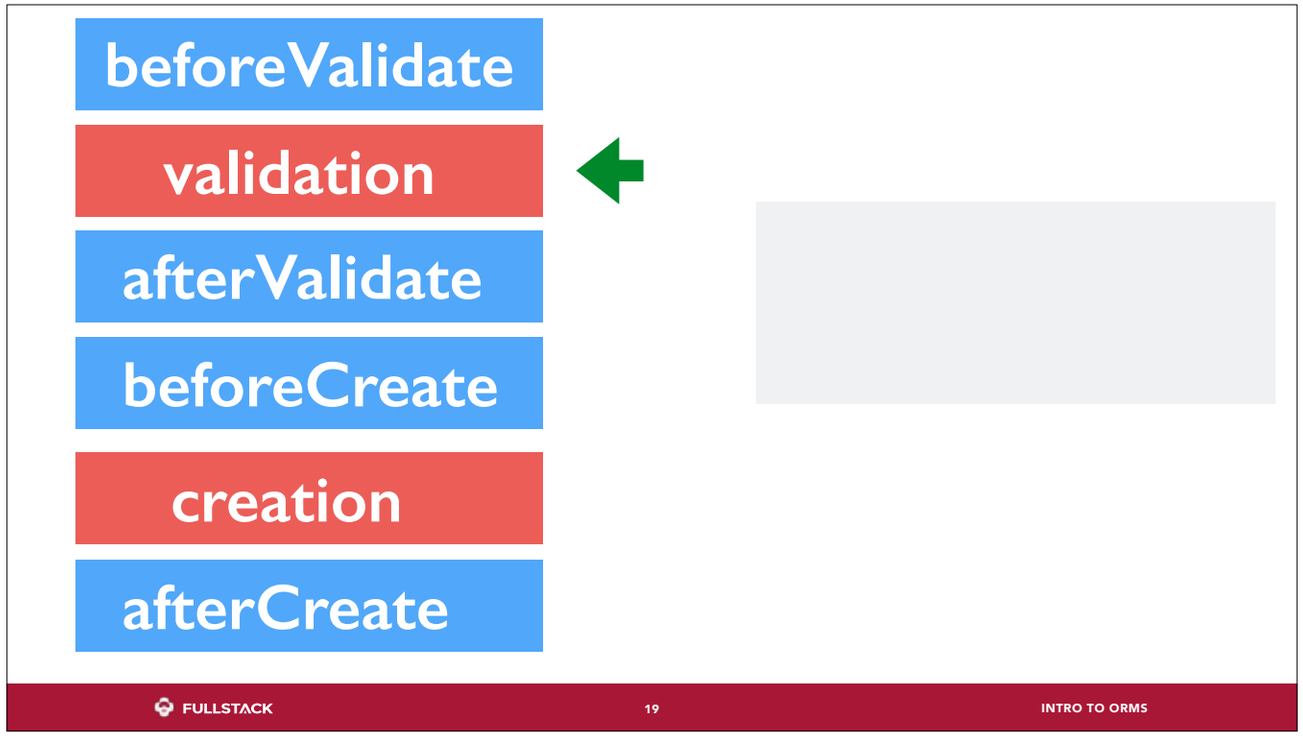
afterValidate

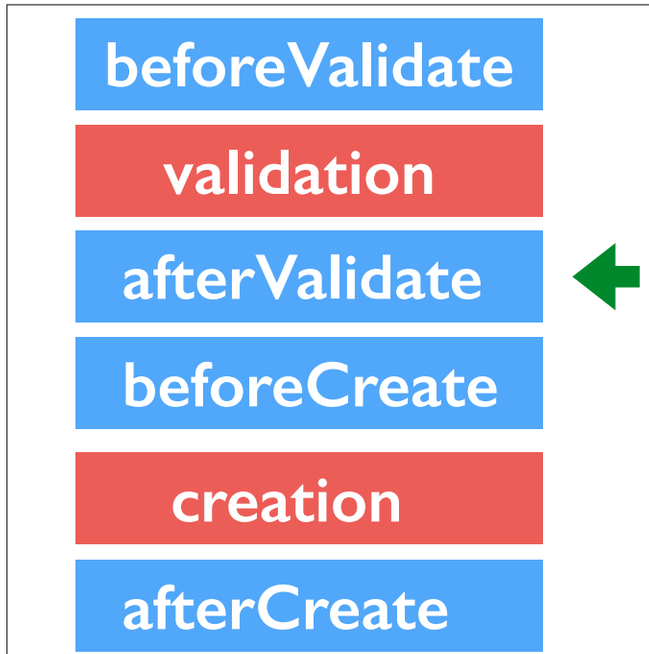
beforeCreate

creation

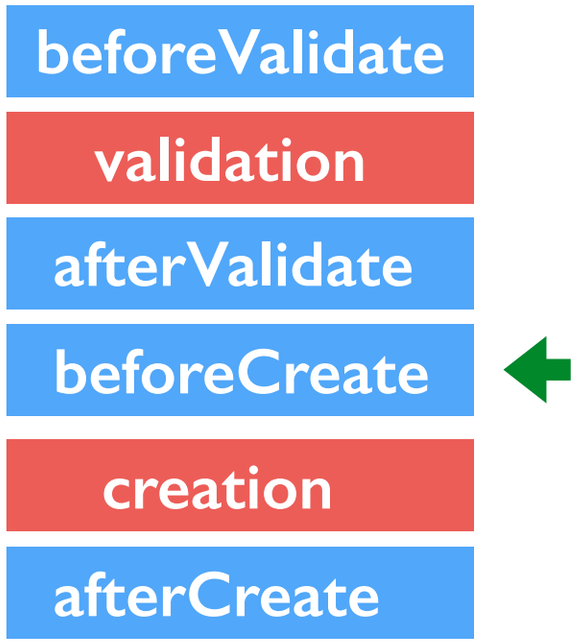
afterCreate

```
User.beforeValidate((user) => {

})
```


beforeValidate
validation
afterValidate
beforeCreate
creation
afterCreate




```
User.afterValidate((user) => {

})
```




```
User.beforeCreate((user) => {

})
```

beforeValidate

validation

afterValidate

beforeCreate

creation

afterCreate

beforeValidate

validation

afterValidate

beforeCreate

creation

afterCreate

```
User.afterCreate((user) => {

})
```

Associations

Associations

- **Establishes** a **relationship** between **two tables**
  (using a foreign-key or a join-table)
- **Creates** several **special** **instance methods** **(like** `getAssociation` **&** `setAssociation`**),** that an **instance** can use to **search for the instances that they are related to.**
- **And more… (eager loading, etc)**

Associations

```
const User = db.define("user", {…})
const Pet  = db.define(“pet", {…})

Pet.belongsTo(User)
User.hasMany(Pet)
```

Associations

```
const User = db.define("user", {…})
const Pet  = db.define(“pet", {…})

Pet.belongsTo(User)
User.hasMany(Pet)
```

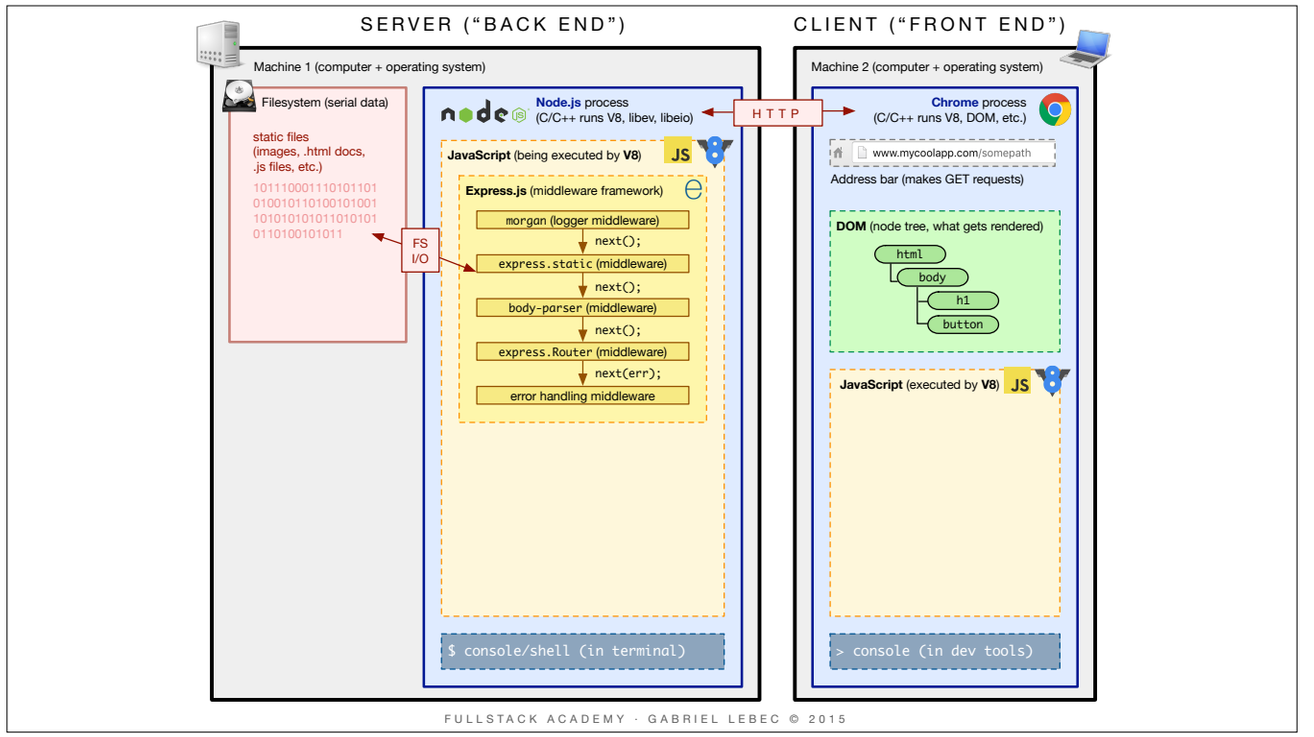
```
const someUser   =  await User.findById(12)
const andHisPet  =  await someUser.getPets()
```

A little more context

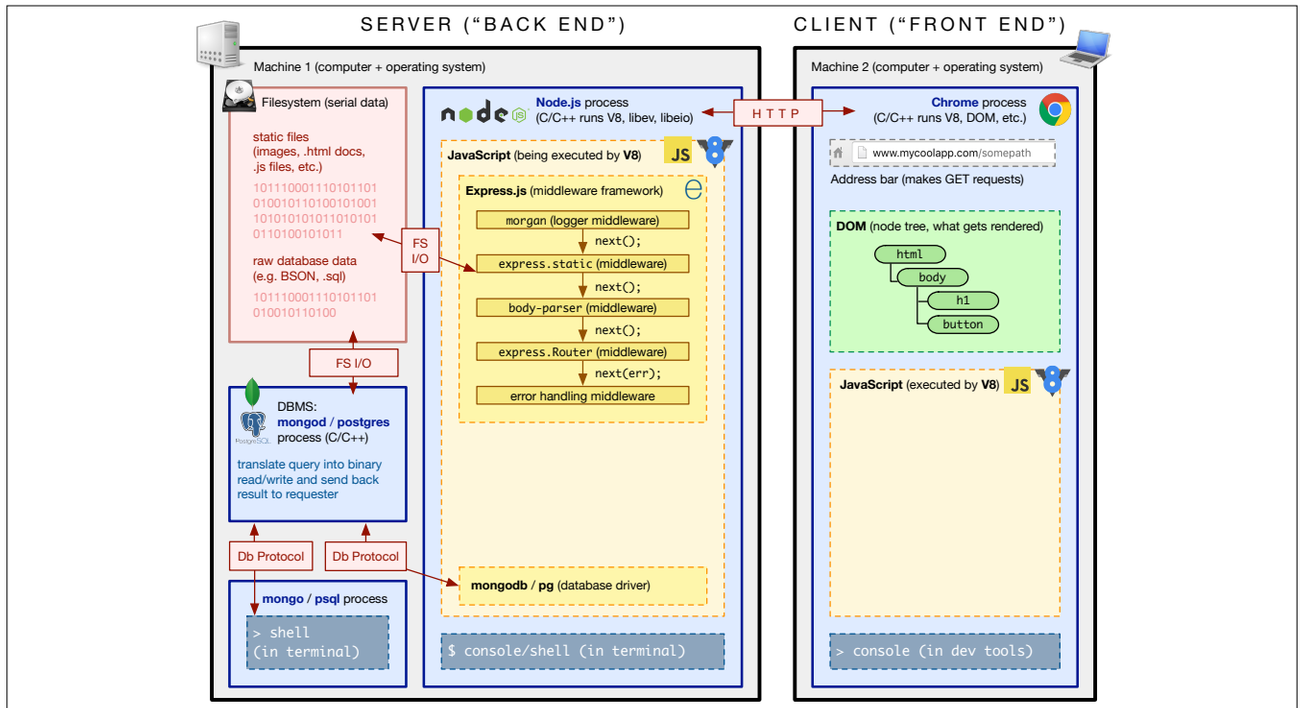
Sequelize

- Lives inside Node.js process
- Knows how to communicate to a few SQL DBMSs, including PostgreSQL and sqlite3

You have to tell it that it is connecting to postgres, which you do in the connection string ‘postgres://'



These are all the technologies we used in the first part of Wizard News



FULLSTACK ACADEMY · GABRIEL LEBEC © 2015

Postgres is a DMBS, a protocol, and a dialect

TL;DR — postgres is a database on our hard-drive that we communicate with via pg in node and it communicates via a dialect of SQL

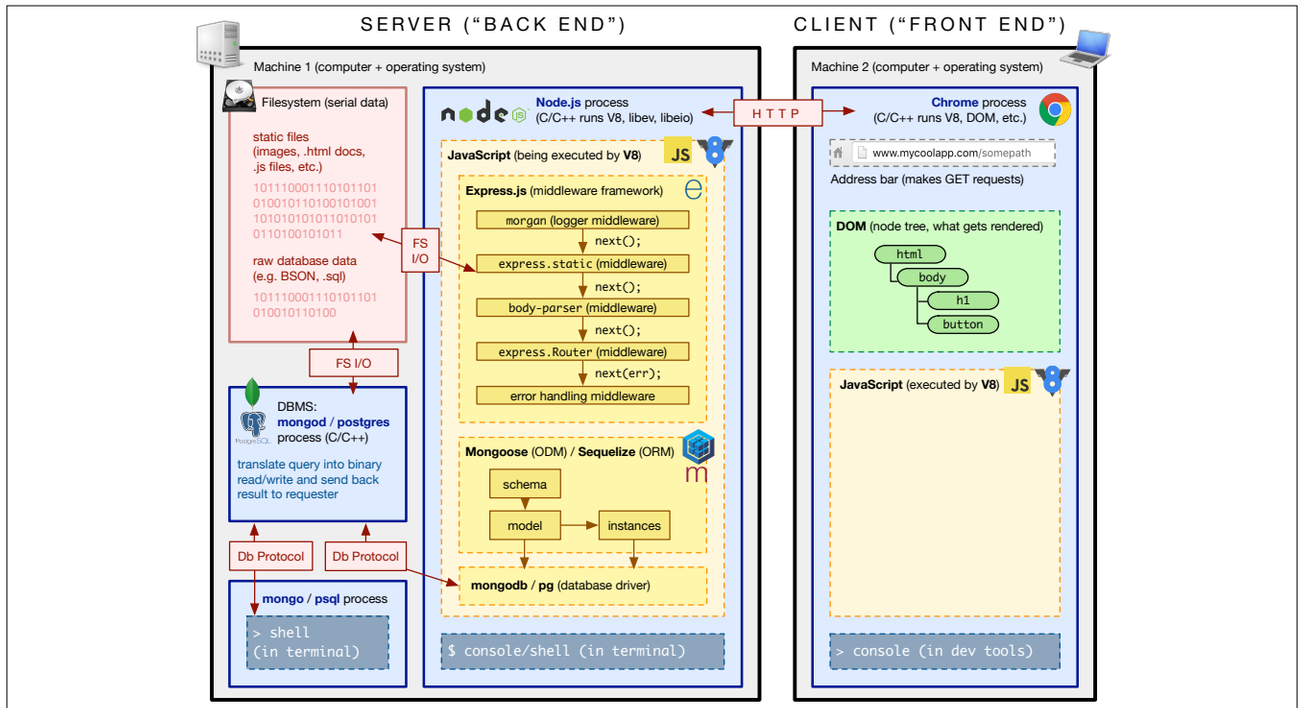
Postgres **the process** is a DBMS — a running program, listening on a TCP port, which knows how to store and retrieve data on disk really well. This is what people typically think of when they say “Postgres is a database” or “we store this in Postgres.”

Postgres “listens on a TCP port,” i.e. it’s a server. However it’s not an HTTP server — the messages aren’t formatted like HTTP messages — it’s a postgres server, i.e. the messages have to be formatted and responded to according to a bunch of arbitrary rules the Postgres people authored. This is why the connection URL for a Postgres DBMS starts with `postgres://` instead of `http://`. Hence, “Postgres” is also a **protocol**.

Instead of implementing the communication with Postgres based on the specification is expects, we use a library to do that for us — `pg`, the “database driver.” So you don’t have to worry about this aspect of postgres.

So we have the **Postgres database** (/ DBMS) listening on a TCP port to communicate messages via the **postgres protocol** (which you don’t have to worry about). What is the actual content of those messages? The **Postgres dialect** of SQL.

A *dialect* in general terms is a variation of a language. SQL — “Structured Query Language” — is the way we typically talk to most relational databases. `SELECT * FROM users;` is valid SQL and the same for a MySQL, SQLite, and Postgres database.



Now Sequelize. Our ORM — Object-Relational Mapper. Sequelize knows zero about the data on our disk. It’s not a database system, it is an abstract way of communicating with the database with objects, translating and passing them to `pg`, getting raw row results back and *transforming rows into full JS objects with methods* — i.e. it maps relations to objects = object-relational mapper.

What’s the point? The point is that arrays of raw row strings are hard to use in a JS program, but arrays of objects with methods like `.save` or `.update` are much easier and natural to use in JS. Sequelize handles that transformation and a lot of other boilerplate DB setup (e.g. creating and naming tables, handling validations, etc.).

Wikistack

- Build a Wikipedia clone
- Walk you through installing and using sequelize
- Application of everything we've learned so far