ORMs part 2

Getting fancy with ORM features

Promises and async/await

Promises and async/await

- Sequelize is a “promise-based” tool
- Instead of plain promises and `.then()`, you can use `async/await` syntax
- Has less curly braces and usually appears less confusing
- You can use normal `if/else` and `try/catch` blocks!

Promises vs. async/await

With promises

```
User.findByPk(req.body.user_id).then(function(user) {
  if (user) {
    return user.sendMessage(req.body.text);
  } else {
    res.send({ status: 'error', error: 'Not found' });
  }
}).then(message => {
  res.send({ status: 'ok', message: message });
}).catch(error => {
  res.send({ status: 'error', error: error });
})
```

With async/await

```
try {
  const user = await User.findByPk(req.body.user_id);
  if (user) {
    const message = await user.send(req.body.text);
    res.send({ status: 'ok', message: message });
  } else {
    res.send({ status: 'error', error: 'Not found' });
  }
} catch (error) {
  res.send({ status: 'error', error: error });
}
```

Querying with ORMs

Querying with ORMs

- ORM models can be queried just like database tables
- To read data, instead of `SELECT` we pass arguments to JavaScript methods like `User.findAll()`
- To write data, instead of `INSERT` we use methods like `User.create()`
- Instead of a table with values in rows and columns, the query returns an array of JavaScript model objects (or just a single JavaScript object)

Querying with and without ORMs

Without ORM

```
SELECT id, name, email
FROM users
WHERE id = 2;
```

With ORM

```
const user = await User.findByPk(2);
console.log(user.name);
```

Querying with and without ORMs

Without ORM

```
SELECT id, name, email
FROM users
WHERE state = 'NY';
```

With ORM

```
const users = await User.findAll({
 where: {
   state: 'NY'
 }
});

for (user in users) {
  ...
}
```

Querying with and without ORMs

Without ORM

```
SELECT id, name, email
FROM users
WHERE email LIKE '%@gmail.com';
```

With ORM

```
const Op = Sequelize.Op;

User.findAll({
 where: {
   email: {
     [Op.like]: '%@gmail.com'
   }
 }
}).then(users => ...);
```

Model methods

Model methods

- ORM models return “active” or “enhanced” JavaScript objects
- Each object represents one table row, but these objects can have other properties not in the database *and even methods*
- This is called the **active record** pattern

Model methods

```
const User = db.define('user', {
  name: Sequelize.STRING,
  email: { type: Sequelize.STRING, unique: true, allowNull: false },
});
User.prototype.sendMessage = function(text) {
  return Message.create({ user_id: this.id, text: text });
};
const user = await User.findByPk(1);
user.sendMessage('hello');
```

Model methods

- Prefer putting core functionality that has to do with your ***data*** or ***domain*** into model methods
- **Views** have the responsibility of receiving and answering client requests
- **Models** have the responsibility of manipulating and fetching the data
- Code that manages data is sometimes called **business logic** - this belongs in your models (or at least avoid putting too much of it in views)

Model validation

Model validation

- Models often should include JavaScript that “checks” the data for special requirements
- These are usually things that SQL `CONSTRAINT` logic can’t guarantee
- It’s better to guarantee things with database constraints when you can

Model validation

http://docs.sequelizejs.com/manual/models-definition.html#validations

Model validation

```
const User = db.define('user', {
  name: Sequelize.STRING,
  email: {
    type: Sequelize.STRING,
    unique: true,
    allowNull: false,
    validate: { isEmail: true }
  }
});
```

Activity: Chatroom models

Homework

- **Reading**: http://docs.sequelizejs.com/manual/migrations.html
- **Assignment**: Capstone data diagram & data model!