Express & Sequelize

Rounding Out

The title of this lecture is Express & Sequelize Rounding Out…. But I don’t think this is a representative title of what we will be doing here, so I prefer to call it…

Express & Sequelize

An assortment of tips, tricks, time savers and conversation starters...

What we will cover

- **Express**
  - Custom error handler
  - 404 Not Found
  - Template engines
- **Sequelize**
  - Eager Loading
  - Class methods / instance methods
  - Many-Many Relationships

Express: Custom Error Handler

Express: Custom Error Handler

Express comes with a built-in error handler

```
router.get("/", async (req, res, next) => {
  try {
    const users = await User.findAll();
    res.send(userList(users));
  } catch (error) { next(error) }
});
```


Express comes with a built-in error handler: We have been using this in our workshops - it’s the one that kicks in when you `next` an error

Express: Custom Error Handler

Express comes with a built-in error handler

```
router.get("/", async (req, res, next) => {
  try {
    const users = await User.findAll();
    res.send(userList(users));
  } catch (error) { next(error) }
});
```

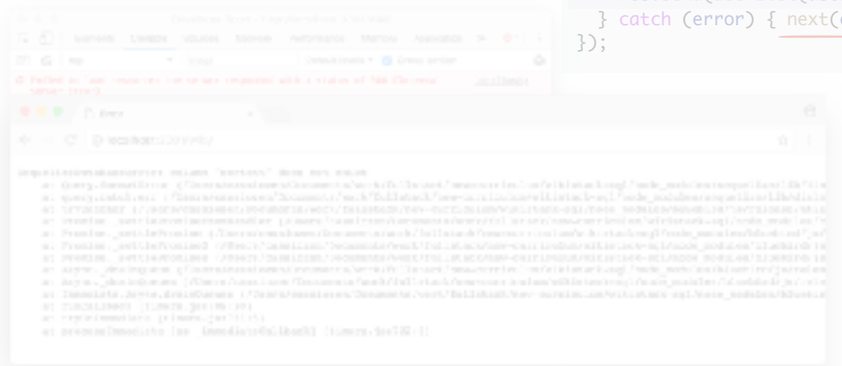
Express comes with a built-in error handler: We have been using this in our workshops - it’s the one that kicks in when you `next` an error

But it has some shortcomings…: It’s not beautiful, it exposes the stack trace, which could include sensible information.

Express: Custom Error Handler

Express comes with a built-in error handler

```
router.get("/", async (req, res, next) => {
  try {
    const users = await User.findAll();
    res.send(userList(users));
  } catch (error) { next(error) }
});
```



Real products you most likely want to show a nice, user-friendly error page.
Or maybe even give some details to the developer in a way that doesn't expose you to hackers (youtube example)…

Express: Custom Error Handler

Real products you most likely want to show a nice, user-friendly error page.
Or maybe even give some details to the developer in a way that doesn't expose you to hackers (youtube example)…

Express: Custom Error Handler

- Define error-handling middleware just like **other middleware**
- except **error-handling functions have four arguments instead of three:** (err, req, res, next)

```
app.use((err, req, res, next) => {
  console.error(err.stack)
  res.status(500)
    .send(/* Some friendly content */)
})
```


And that's how you make a custom error handling middleware.

Moving on…

Express: 404 Not Found

Express: 404 Not Found

```
...
// All routes and middlewares above

// Last, Handle 404:
app.use((req, res) => {
  res.status(404).send(/*Not found*/);
});
```


Just add a catch-all middleware bellow all other existing, valid routes and middlewares. If no other route matches, the 404 will match.

Moving on…

Express: Template Engines

Express: Template Engines

- JavaScript template literals ≠ template engine.
- Templates: static text files with special annotations.
- Template engines:
  - Interprets the document;
  - Replaces variables with actual values
  - Send them as HTML in the HTTP response.

Express: Template Engines

userList.js

```
const html = require("html-template-tag");
const layout = require("./layout");

module.exports = (users) => layout(html`
  <h3>Users</h3>
  <hr>
  <ul class="list-unstyled">
    <ul>
      ${users.map(user => html`<li>
        <a href="/users/${user.id}">${user.name}</a>
      </li>`)}
    </ul>
  </ul>
`);
```

userList.html

```
{% extends "layout.html" %}

{% block content %}
<h3>Users</h3>
<hr>
<ul class="list-unstyled">
  <ul>
    {% for user in users %}
    <li>
       <a href="/users/{{ user.id }}">{{ user.name }}</a>
    </li>
    {% endfor %}
  </ul>
</ul>
{% endblock %}
```


This is a direct comparison between the userList view module you guys just created in the wiki stack. On the right we have a similar example using swig/nunjucks. Note a few key-differences:

- The template file is not javascript - it's just a plain html file
- Notice how it contains special annotations for looping and printing values. These are not javascript commands, they are specific for each template engine.

Template engines can be more convenient in some circumstances, but at cost of having to learn the template engine and available syntax.

We are not going to use a template engine in your workshops - We just want you to be aware that such things exists and are fairly common in the `Express` world.

Moving on…

Express: Template Engines

userList.js

```
const html = require("html-template-tag");
const layout = require("./layout");

module.exports = (users) => layout(html`
  <h3>Users</h3>
  <hr>
  <ul class="list-unstyled">
    <ul>
      ${users.map(user => html`<li>
        <a href="/users/${user.id}">${user.name}</a>
      </li>`)}
    </ul>
  </ul>
`);
```

userList.html

```
{% extends "layout.html" %}

{% block content %}
<h3>Users</h3>
<hr>
<ul class="list-unstyled">
  <ul>
    {% for user in users %}
    <li>
       <a href="/users/{{ user.id }}">{{ user.name }}</a>
    </li>
    {% endfor %}
  </ul>
</ul>
{% endblock %}
```


This is a direct comparison between the userList view module you guys just created in the wiki stack. On the right we have a similar example using swig/nunjucks. Note a few key-differences:

- The template file is not javascript - it's just a plain html file
- Notice how it contains special annotations for looping and printing values. These are not javascript commands, they are specific for each template engine.

Template engines can be more convenient in some circumstances, but at cost of having to learn the template engine and available syntax.

We are not going to use a template engine in your workshops - We just want you to be aware that such things exists and are fairly common in the `Express` world.

Moving on…

Sequelize: Eager Loading

Sequelize: not Eager Loading

```
const pages = await Page.findAll();

for (let i = 0; i < pages.length; i++) {
  const author = await pages[i].getAuthor();
}
```


This is valid Sequelize syntax and will work. But… do you see any problems here?

It is perfectly ok to call getAuthor if I had one page, but in this case I'm making one new query for an author for every page - this is super wasteful.

Sequelize: not Eager Loading

```
const pages = await Page.findAll();

for (let i = 0; i < pages.length; i++) {
  const author = pages[i].getAuthor();
}
```


This is valid Sequelize syntax and will work. But… do you see any problems here?

It is perfectly ok to call getAuthor if I had one page, but in this case I'm making one new query for an author for every page - this is super wasteful.

Sequelize: Eager Loading

- In raw SQL queries, we have INNER JOIN
- In Sequelize - it just goes by the name of “eager loading”
- Don't get hung up on the terminology
  when you see “eager loading”, think “join two tables”.

More formal definition of eager loading: Querying one type of entity while also fetching the related entities as part of the query.

Sequelize: Eager Loading

```
const pages = await Page.findAll({
  include: [
    {model: User, as: 'author'}
  ]
});
```

Sequelize: Class & instance methods

Sequelize: Class & instance methods

This is Sequelize's source code for the model's `define` method. Notice that a Model is just a class, which in turn is just sugar for a constructor function.

This means we can add our own functions to the constructor function OR to its prototype.

Sequelize: Class & instance methods

- Common/Convenient ways to add functionality to your Sequelize models

This is Sequelize's source code for the model's `define` method. Notice that a Model is just a class, which in turn is just sugar for a constructor function.

This means we can add our own functions to the constructor function OR to its prototype.

Sequelize: Class & instance methods

- Common/Convenient ways to add functionality to your Sequelize models

This is Sequelize's source code for the model's `define` method. Notice that a Model is just a class, which in turn is just sugar for a constructor function.

This means we can add our own functions to the constructor function OR to its prototype.

Sequelize: Class & instance methods

- **Common/Convenient ways to add functionality to your Sequelize models**
- It's just Javascript:
  We can add functions to the constructor function OR to its prototype.

This is Sequelize's source code for the model's `define` method. Notice that a Model is just a class, which in turn is just sugar for a constructor function.

This means we can add our own functions to the constructor function OR to its prototype.

Sequelize: Class & instance methods

```
const Dog = db.define('dog', {/* etc*/})
```


The code example below demonstrates a class method. Class methods are methods that are available on the model itself (aka the class). We often write these to get instances, or do something to more than one instance.

Sequelize: Class & instance methods

```
const Dog = db.define('dog', {/* etc*/})

Dog.findPuppies = function () {
  // 'this' refers directly back to the model
  return this.findAll({ // Dog.findAll
    where: {
      age: {$lte: 1}
    }
  })
}
```


The code example below demonstrates a class method. Class methods are methods that are available on the model itself (aka the class). We often write these to get instances, or do something to more than one instance.

Sequelize: Class & instance methods

```
const Dog = db.define('dog', {/* etc*/})

Dog.findPuppies = function () {
  // 'this' refers directly back to the model
  return this.findAll({ // Dog.findAll
    where: {
      age: {$lte: 1}
    }
  })
}
```

Sequelize: Class & instance methods

```
const Dog = db.define('dog', {/* etc*/})

Dog.findPuppies = function () {
  // 'this' refers directly back to the model
  return this.findAll({ // Dog.findAll
    where: {
      age: {$lte: 1}
    }
  })
}
```

In your routes, for example…

```
const foundPuppies = await Dog.findPuppies()
```

Sequelize: Class & instance methods

```
const Dog = db.define('dog', {/* etc*/})
```


The code example below demonstrates an instance method. Instance methods are methods that are available on instances of the model. We often write these to get information or do something related to that instance.

Sequelize: Class & instance methods

```
const Dog = db.define('dog', {/* etc*/})

Dog.prototype.isBirthday = function () {
  const today = new Date()
  // 'this' refers to the instance itself
  if (this.birthDay === today.getDate()) {
    return true;
  } else {
    return false;
  }
}
```


The code example below demonstrates an instance method. Instance methods are methods that are available on instances of the model. We often write these to get information or do something related to that instance.

Sequelize: Class & instance methods

```
const Dog = db.define('dog', {/* etc*/})

Dog.prototype.isBirthday = function () {
  const today = new Date()
  // 'this' refers to the instance itself
  if (this.birthDay === today.getDate()) {
    return true;
  } else {
    return false;
  }
}
```

Sequelize: Class & instance methods

```
const Dog = db.define('dog', {/* etc*/})

Dog.prototype.isBirthday = function () {
  const today = new Date()
  // 'this' refers to the instance itself
  if (this.birthDay === today.getDate()) {
    return true;
  } else {
    return false;
  }
}
```

In your routes, for example...

```
const createdDog = new Dog({name: 'Pork Chop'})

// the instance method is invoked *on the instance*
if(createdDog.isBirthday) console.log("Happy birthday!")
```

Sequelize: Class & instance methods

Don’t go crazy about how SEQUELIZE provides class methods and instance methods.
it’s just JavaScript: You can attach your own methods to any constructor function or it’s prototypes.

Sequelize: Class & instance methods

Don’t go crazy about how SEQUELIZE provides class methods and instance methods.
it’s just JavaScript: You can attach your own methods to any constructor function or it’s prototypes.

Sequelize: Many-Many Relationships

ONE-ONE

```
Pug.belongsTo(Owner)
Owner.hasOne(Pug)
```

PUGS

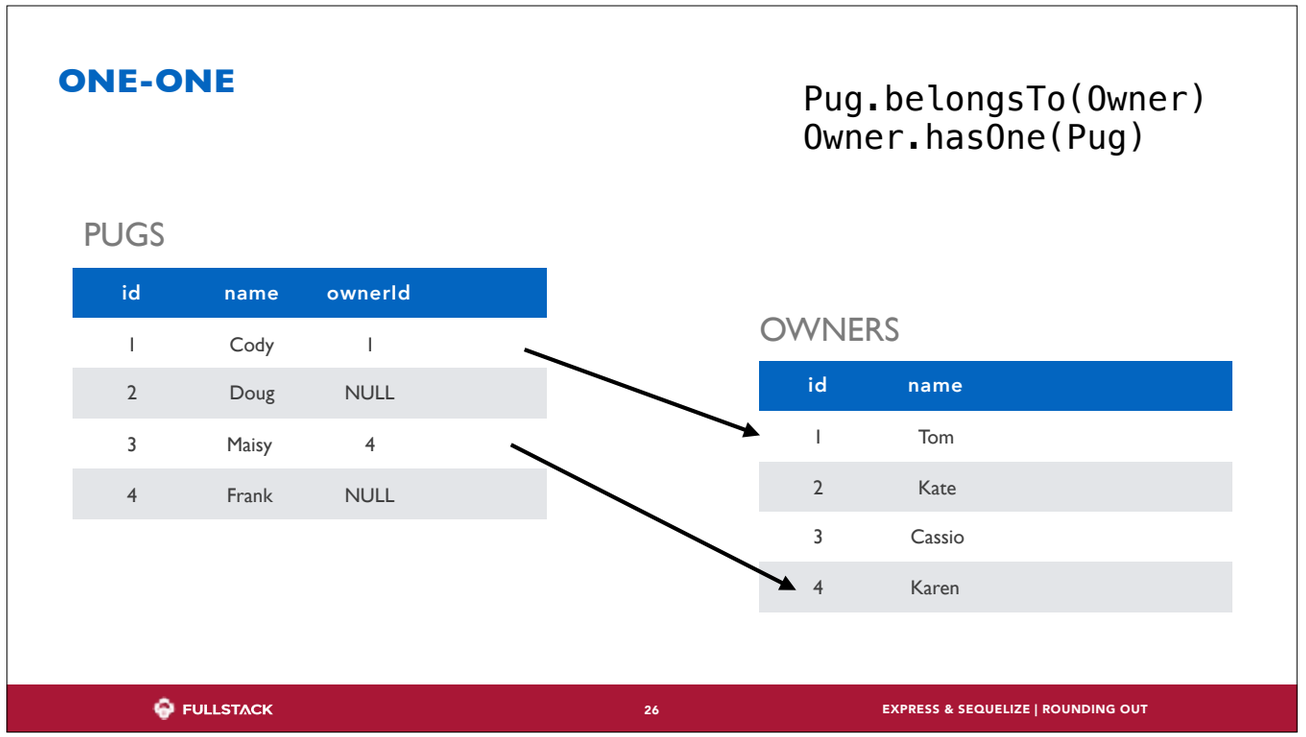
id	name	ownerId
1	Cody	1
2	Doug	NULL
3	Maisy	4
4	Frank	NULL

OWNERS

id	name
1	Tom
2	Kate
3	Cassio
4	Karen

Let’s review - here’s a one to many relationship. In Sequelize, it’s defined by invoking the “belongsTo” method on the model whose table should contain the foreign key, and the “hasOne” on the model that the foreign key should point to.

In this example, we have a table called Pugs and a table called Owners. Pugs “belong to” an Owner, and an owner has one Pug. This means that in the pugs table, there will be a foreign key table called “ownerId”, which will point to a row in the owners table. Because this is a one-one relationship, we know that the values for the ownerId column in the pugs table will point to that pug’s unique owner. This also means that the association methods on the owner will be in the singular - like “getPug” and “setPug”



Let’s review - here’s a one to many relationship. In Sequelize, it’s defined by invoking the “belongsTo” method on the model whose table should contain the foreign key, and the “hasOne” on the model that the foreign key should point to.

In this example, we have a table called Pugs and a table called Owners. Pugs “belong to” an Owner, and an owner has one Pug. This means that in the pugs table, there will be a foreign key table called “ownerId”, which will point to a row in the owners table. Because this is a one-one relationship, we know that the values for the ownerId column in the pugs table will point to that pug’s unique owner. This also means that the association methods on the owner will be in the singular - like “getPug” and “setPug”

ONE-MANY

```
Pug.belongsTo(Owner)
Owner.hasMany(Pug)
```

PUGS

id	name	ownerId
1	Cody	1
2	Doug	1
3	Milo	2
4	Frank	NULL

OWNERS

id	name
1	Tom
2	Kate
3	Cassio
4	Karen

If an owner can have multiple pugs, we might specify that an owner “has many” pugs. This means that we expect that multiple pugs could have the same value for ownerId. This also means that the association methods on the owner will be plural - like “getPugs”, “setPugs” and “addPugs”.

ONE-MANY

```
Pug.belongsTo(Owner)
Owner.hasMany(Pug)
```

PUGS

id	name	ownerId
1	Cody	1
2	Doug	1
3	Milo	2
4	Frank	NULL

OWNERS

id	name
1	Tom
2	Kate
3	Cassio
4	Karen

If an owner can have multiple pugs, we might specify that an owner “has many” pugs. This means that we expect that multiple pugs could have the same value for ownerId. This also means that the association methods on the owner will be plural - like “getPugs”, “setPugs” and “addPugs”.

MANY-MANY

```
Pug.belongsToMany(Friend, {through: “pug_friend”})
Friend.belongsToMany(Pug, {through: “pug_friend”})
```

PUG_FRIEND

pug_id	friend_id
1	1
1	2
2	1

PUGS

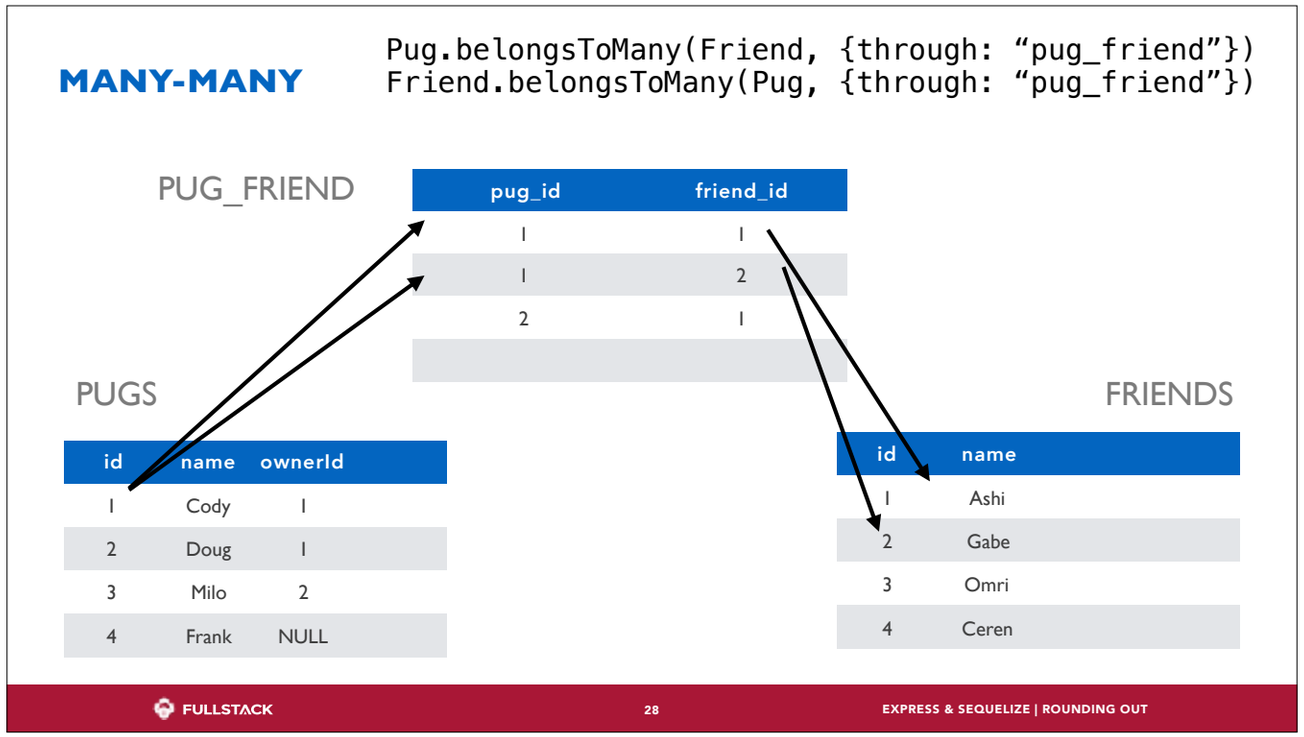
id	name	ownerId
1	Cody	1
2	Doug	1
3	Milo	2
4	Frank	NULL

FRIENDS

id	name
1	Ashi
2	Gabe
3	Omri
4	Ceren

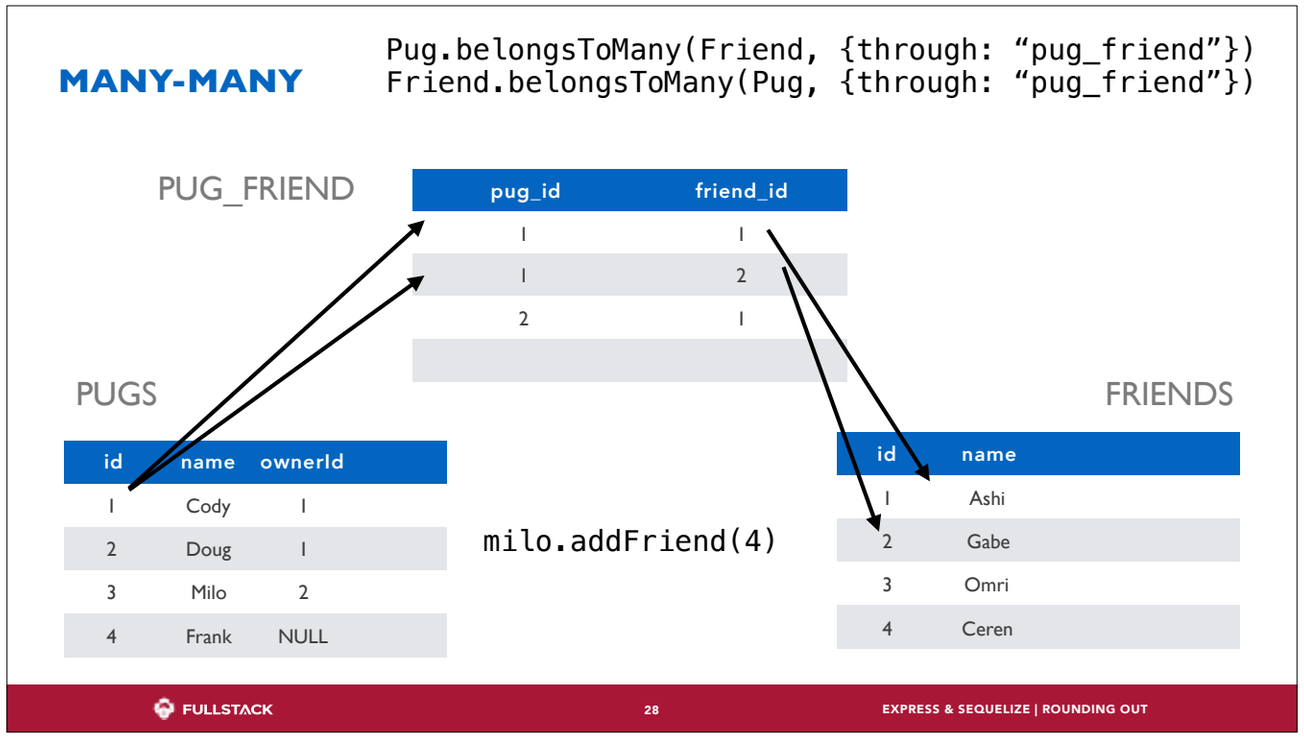
So a pug only has one owner, but a pug could have many friends. And if someone is friends with pugs, it’s likely that they’re friends with more than one! This is the nature of a many-many relationship. In a many-many relationship, we have an intermediary “join” table that represents this relationship. For example, if we want to find all of Cody’s friends, we look up all of the rows in the “pug_friend” table with the pug_id of 1, and we see that there are rows with friend_ids of 1 and 2 - so Cody is friends with both Ashi and Gabe.

In Sequelize, we can define a many to many relationship by using the “belongsToMany” method. We typically define this on both models. We say, “a pug belongs to many friends, through the join table pug_friend. Likewise, a friend belongs to many pugs, through the table pug_friend.” By doing this, Sequelize automatically creates the “pug_friend” join table, and gives us association methods that will easily populate it. For example, if we have a pug instance and we say “pug.addFriend()”, this will create a new row in the join table. This would also work if we did “friend.addPug”.



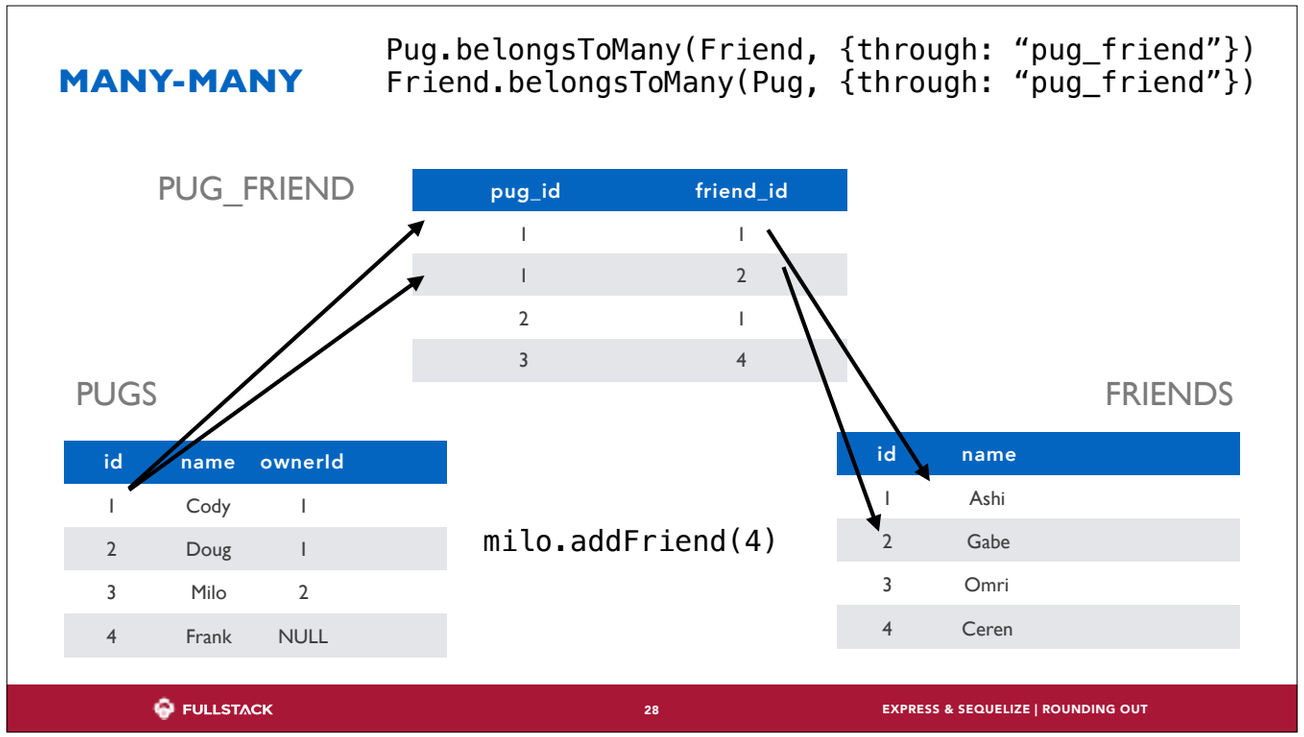
So a pug only has one owner, but a pug could have many friends. And if someone is friends with pugs, it's likely that they're friends with more than one! This is the nature of a many-many relationship. In a many-many relationship, we have an intermediary "join" table that represents this relationship. For example, if we want to find all of Cody's friends, we look up all of the rows in the "pug_friend" table with the pug_id of 1, and we see that there are rows with friend_ids of 1 and 2 - so Cody is friends with both Ashi and Gabe.

In Sequelize, we can define a many to many relationship by using the "belongsToMany" method. We typically define this on both models. We say, "a pug belongs to many friends, through the join table pug_friend. Likewise, a friend belongs to many pugs, through the table pug_friend." By doing this, Sequelize automatically creates the "pug_friend" join table, and gives us association methods that will easily populate it. For example, if we have a pug instance and we say "pug.addFriend()", this will create a new row in the join table. This would also work if we did "friend.addPug".



So a pug only has one owner, but a pug could have many friends. And if someone is friends with pugs, it’s likely that they’re friends with more than one! This is the nature of a many-many relationship. In a many-many relationship, we have an intermediary “join” table that represents this relationship. For example, if we want to find all of Cody’s friends, we look up all of the rows in the “pug_friend” table with the pug_id of 1, and we see that there are rows with friend_ids of 1 and 2 - so Cody is friends with both Ashi and Gabe.

In Sequelize, we can define a many to many relationship by using the “belongsToMany” method. We typically define this on both models. We say, “a pug belongs to many friends, through the join table pug_friend. Likewise, a friend belongs to many pugs, through the table pug_friend.” By doing this, Sequelize automatically creates the “pug_friend” join table, and gives us association methods that will easily populate it. For example, if we have a pug instance and we say “pug.addFriend()”, this will create a new row in the join table. This would also work if we did “friend.addPug”.



So a pug only has one owner, but a pug could have many friends. And if someone is friends with pugs, it's likely that they're friends with more than one! This is the nature of a many-many relationship. In a many-many relationship, we have an intermediary "join" table that represents this relationship. For example, if we want to find all of Cody's friends, we look up all of the rows in the "pug_friend" table with the pug_id of 1, and we see that there are rows with friend_ids of 1 and 2 - so Cody is friends with both Ashi and Gabe.

In Sequelize, we can define a many to many relationship by using the "belongsToMany" method. We typically define this on both models. We say, "a pug belongs to many friends, through the join table pug_friend. Likewise, a friend belongs to many pugs, through the table pug_friend." By doing this, Sequelize automatically creates the "pug_friend" join table, and gives us association methods that will easily populate it. For example, if we have a pug instance and we say "pug.addFriend()", this will create a new row in the join table. This would also work if we did "friend.addPug".

Workshop

- **Express**
  - Custom error handler
  - 404 Not Found
  - Template engines
- **Sequelize**
  - Eager Loading
  - Class methods / instance methods
  - Many-Many relationships

We’re not going to use template engines or many-many for the workshop, but we are going to use all the rest.