Galvanize Bookshelf Hints

Objectives

- Understand how to translate postgres fields to JSON API format
- Understand how to camelize and decamelize object properties
- Know how to use Boom node module to throw errors
- Be able to reset a postgres id sequence with knex

Which of these two JSON formats is correct?

Format 1


```
{
    "id": 1,
    "name": "Bootsy",
    "age": 3,
    "skill": "birding",
    "description": "Calico Cat",
    "profile_url": "meow.png",
    "created_at": "2016-11-15T16:00:55.104Z",
    "updated_at": "2016-11-15T16:00:55.104Z"
}
```

Format 2


```
{
    "id": 1,
    "name": "Bootsy",
    "age": 3,
    "skill": "birding",
    "description": "Calico Cat",
    "profileUrl": "meow.png",
    "createdAt": "2016-11-15T16:00:55.104Z",
    "updatedAt": "2016-11-15T16:00:55.104Z"
}
```

From the Google JSON Style Guide

Property Names should...

- Be meaningful names with defined semantics.
- Must be camel-cased, ascii strings.
- The first character must be a letter, an underscore (_) or a dollar sign ($).
- Subsequent characters can be a letter, a digit, an underscore, or a dollar sign.
- Reserved JavaScript keywords should be avoided.

What do you do if you used an underscore naming convention for your database fields?

When we send things out from our database we need to translate:
profile_url => profileUrl, created_at => createdAt, etc.

When we recieve requests we need to go the other way:
profileUrl => profile_url

camelizeKeys and decamelizeKeys to the rescue!

```
const { camelizeKeys, decamelizeKeys } = require('humps');

camelizeKeys({hello_world: 'howdy'})    // {helloWorld: 'howdy'}
decamelizeKeys({theCats: 'meow'})       // {the_cats: 'meow'}
```

Throwing Errors with Node

```
router.post('/', (req, res, next) => {

  const { name, skills, description, profileUrl } = req.body;

  if (!name || !name.trim()) {
    res.status(400).send("Name must not be blank");
    return;
  }

  // more cool code ...
}
```

Throwing Errors with Boom

```
const boom = require('boom')

router.post('/', (req, res, next) => {

  const { name, skills, description, profileUrl } = req.body;

  if (!name || !name.trim()) {
    next(boom.create(400, 'Name must not be blank.'));
    return;
  }

  // more cool code ...
}
```

What are some advantages to the boom approach?

- centralized error logging
- ability to handle errors differently in development vs. production
- not "rollling your own" in your routers (which is error prone)

If you import data with set ids in postgres, it will get the sequence out of whack. Then when you try to insert new data it will says there is a primary key error.

```
‘ERROR:  duplicate key value violates unique constraint “books_pkey” ’
```

Questions:

- What does a sequence do for a serial id ?
- What method method in knex allows you to execute sql commands directly

This would be an example of how to reset a sequence for a table called `books` after seeding it.

```
.then(() => {
  return knex.raw(
    "SELECT setval('books_id_seq', (SELECT MAX(id) FROM books));"
  );
});
```

References:

https://github.com/domchristie/humps

https://google.github.io/styleguide/jsoncstyleguide.xml#Property*Name*Format

https://github.com/hapijs/boom

https://www.postgresql.org/docs/9.6/static/functions-sequence.html

http://knexjs.org/#Schema-raw